



**ANDROID PROTECTION SYSTEM: A SIGNED CODE SECURITY
MECHANISM FOR SMARTPHONE APPLICATIONS**

THESIS

Jonathan D. Stueckle, Capt, USAF

AFIT/GCE/ENG/11-06

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. government and is not subject to copyright protection in the United States.

ANDROID PROTECTION SYSTEM: A SIGNED CODE SECURITY
MECHANISM FOR SMARTPHONE APPLICATIONS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Jonathan D. Stueckle, B.S. Computer Engineering

Captain, USAF

March 2011

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ANDROID PROTECTION SYSTEM: A SIGNED CODE SECURITY
MECHANISM FOR SMARTPHONE APPLICATIONS

Jonathan D. Stueckle, B.S. Computer Engineering

Captain, USAF

Approved:



Dr. Rusty O. Baldwin (Chairman)

3-3-11

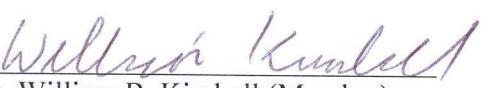
Date



Dr. Richard A. Raines (Member)

4 Mar 11

Date



Mr. William B. Kimball (Member)

3/4/11

Date

Abstract

This research develops the Android Protection System (APS), a hardware-implemented application security mechanism on Android smartphones. APS uses a hash-based white-list approach to protect mobile devices from unapproved application execution. Functional testing confirms this implementation allows approved content to execute on the mobile device while blocking unapproved content. Performance benchmarking shows system overhead during application installation increases linearly as the application package size increases. APS presents no noticeable performance degradation during application execution. The security mechanism degrades system performance only during application installation, when users expect delay.

APS is implemented within the default Android application installation process. Applications are hashed prior to installation and compared against a white-list of approved content. APS allows applications that generate a matching hash; all others are blocked. APS blocks 100% of unapproved content while allowing 100% of approved content. Performance overhead for APS varies from 100.5% to 112.5% with respect to the default Android application installation process. This research directly supports the efforts of the USAF and the DoD to protect our information and ensure that adversaries do not gain access to our systems.

Acknowledgements

I would like to thank my advisor and committee for their support and contribution to this work. It would not have been possible without their suggestions and guidance along the way. I am grateful to my advisor, Dr. Rusty Baldwin, for providing excellent classroom instruction as well as expert guidance in selecting a topic, narrowing research area, and staying on task. I thank my committee members, Dr. Richard Raines and Mr. William Kimball for their expertise and guidance throughout the research process. Most importantly, I thank my wife for her constant support and encouragement. I deeply appreciate the sacrifices she made as I spent countless hours away from home. She is probably more excited than I am to have this work completed. Thank you.

Table of Contents

	Page
Abstract	iv
Acknowledgements	v
Table of Contents	vi
List of Figures	viii
List of Tables.....	ix
I. Introduction.....	1
1.1 Research Domain.....	1
1.2 Problem Statement.....	2
1.3 Research Goals	2
1.4 Document Outline	3
II. Literature Review	4
2.1 Introduction to ARM Architecture	4
2.1.1 Programmers' Model.....	4
2.1.2 Instruction Set.....	9
2.1.3 Addressing Modes	13
2.1.4 Memory and System Architectures	15
2.1.5 Vector Floating-Point Architecture	17
2.2 Introduction to Google Android Operating System	17
2.2.1 Operating System Background.....	18
2.2.2 Advantages Offered by Android.....	19
2.2.3 Known Implementations of Operating System.....	20
2.2.4 Summary.....	22
2.3 Examination of Android Protection Mechanisms	23
2.3.1 Component Interactions.....	23
2.3.2 Built-in Security Features.....	23
2.3.3 Current State of Protection Mechanisms	25
2.3.4 Proposal for Improved Security Measures	30
2.3.5 Summary.....	30
III. Methodology	32

3.1	Background	32
3.2	Problem Definition	32
3.2.1	Goals and Hypothesis	33
3.2.2	Approach	33
3.3	System Boundaries	34
3.4	System Services.....	35
3.5	Workload	36
3.6	Performance Metrics	36
3.7	System Parameters.....	37
3.8	Factors	39
3.9	Evaluation Technique	41
3.10	Experimental Design	42
3.11	Methodology Summary	42
IV.	Android Protection System Performance	44
4.1	Introduction	44
4.2	Android Protection System Implementation	44
4.2.1	White-list Creation	45
4.2.2	Hash Digest Placement.....	45
4.3	Evaluation Technique	46
4.4	Functional Protection Testing.....	47
4.5	Performance Benchmark	49
4.6	Summary	54
V.	Conclusions	55
5.1	Research Accomplishments.....	55
5.2	Research Impact	56
5.3	Future Research Areas.....	57
Appendix A.	MD5 Message Digest Algorithm	59
Appendix B.	APS Modification to Android OS 1.5	73
Bibliography	84

List of Figures

Figure	Page
2.1 ARM Register Organization	6
2.2 ARM Instruction Encoding Pattern	10
2.3 Overview of the Android Application Stack	19
2.4 SCanDroid Architecture of Analysis	26
3.1 Android Protection System	35
4.1 Application Load Time – APS Enabled.....	51
4.2 Application Load Time – APS Disabled	52
4.3 Difference in Application Load Times	53

List of Tables

Table	Page
2.1 ARM Processor Modes	5
2.2 ARM Exception Types	7
3.1 Experimental Factors	39
4.1 APS Functional Protection Results	48
4.2 APS Performance Testing Results	50
4.3 Difference in Mean Load Times	52

ANDROID PROTECTION SYSTEM: A SIGNED CODE SECURITY MECHANISM FOR SMARTPHONE APPLICATIONS

I. Introduction

1.1 Research Domain

Mobile devices are an ever-increasing part of our society. Cut off network access or service coverage and life for many comes screeching to a halt. Mobile phones expand communication networks beyond wired limitations. Smartphones, mobile phones that can execute third party code, further extend the capabilities now at the fingertips of the general public. Smartphone users have the ability to control home security features, modify house lighting, start cars, manage bank accounts, stream online video, use Global Positioning System (GPS) services, and many more, all from the convenience of a mobile phone application.

Gartner estimates that worldwide mobile phone sales for Q3 2010 totaled 417 million devices, of which 81 million were smartphones. Smartphone sales grew 96 percent from Q3 2009, accounting for 19.3 percent of overall mobile phone sales. Google's Android Operating System accounted for 25.5 percent of smartphone sales, up from 3.5 percent in 2009 [GN10].

As smartphone use increases, the security of applications and underlying code becomes increasingly important. This research focuses on the domain of smartphone application security, specifically on the Android Operating System.

1.2 Problem Statement

Increased smartphone use is a serious security issue. Smartphone applications access and store sensitive information including GPS location, Short Message Service (SMS) billing, bank account login credentials, premium phone calls, e-mails, and text messages. Access to this information greatly incentivizes malicious application developers to create new ways to steal sensitive data.

Anyone can write, sign, and submit an application to the Android Market. Many of these applications are available for free download. The average smartphone in the United States has 22 applications installed [NW10]. However, Android application security depends almost solely on decisions users make when downloading and installing applications. Numerous applications reside on smartphones with no mechanism in place to protect against malicious code execution.

1.3 Research Goals

The existing application security solution on the Android Operating System is inadequate, relying heavily on user discretion. Signature-based mobile phone security (anti-virus, anti-spyware, etc.) is unable to keep up with the rapid growth in smartphone use. Therefore, malicious content is freely available and often infects mobile devices. The goal of this research is to improve application security on a mobile platform.

This research adheres to a defense-in-depth strategy. The native security in the Android Operating System is left intact. This research adds a complementary security mechanism to prevent unauthorized application code from executing on an Android device. The approach implements the security mechanism within kernel space of the

operating system so that the protection itself cannot be compromised by malicious code. Once implemented, the security mechanism is benchmarked for performance overhead and protection effectiveness.

1.4 Document Outline

Chapter II summarizes the ARM architecture, the Android Operating System, and the current state of security mechanisms—those native to Android as well as third-party products. Chapter III introduces the methodology for developing, implementing, and evaluating a new application security mechanism, called Android Protection System (APS). This mechanism ensures that application packages on the Android platform are verified before code is allowed to execute. Chapter IV discusses and analyzes the results from benchmarking the performance of APS. Finally, Chapter V highlights accomplishments of this research, focusing on the impact to the smartphone community as well as suggestions for future work.

II. Literature Review

2.1 Introduction to ARM Architecture

Most mobile phones use a microprocessor based on the ARM architecture [SAH09]; Google also selected ARM as the architecture on which to develop the Android Operating System (OS). This architecture, with processors available to meet a variety of performance, power, area, and application needs, is a natural fit for mobile devices. This section explores five aspects of the ARM architecture and examines Android's use of this architecture. The Programmers' Model and Instruction Set sections examine the architecture in general. The Addressing Modes, Memory and System Architectures, and Vector Floating-Point Architecture sections look more in-depth at the architecture and how they apply to the Android OS.

2.1.1 *Programmers' Model*

The Programmers' Model portion of the ARM Architecture Manual [ARM05] describes various aspects of the architecture including data types, processor modes, registers, general-purpose registers, program status registers, exceptions, endian support, unaligned access support, synchronization primitives, the Jazelle Extension, and saturated integer arithmetic. All of these are crucial to understand application programming for the Android device and for making kernel-level modifications to the system.

ARM supports byte, halfword, and word data types. Most data operations are performed on word quantities [ARM05]. ARM instructions are exactly one word and aligned on four-byte boundaries. The architecture has seven processor modes as shown in Table 2.1. All modes other than user mode are considered privileged modes and are

not accessible other than via an exception. The privileged modes have full access to system resources and can freely change mode.

Table 2.1. ARM Processor Modes [ARM05]

Processor mode	Mode number	Description
User	usr	0b10000
FIQ	fiq	0b10001
IRQ	irq	0b10010
Supervisor	svc	0b10011
Abort	abt	0b10111
Undefined	und	0b11011
System	sys	0b11111

The ARM architecture provides 37 programmer accessible registers, 31 general-purpose registers and six status registers. All registers are 32 bits in width, although the status registers typically do not allocate or implement all 32 bits. The architecture arranges the registers in banks that partially overlap. The current mode determines which of the banks are available. Each processor mode has access to 15 general-purpose registers, one or two status registers, and the program counter. An overview of this layout is shown in Figure 2.1 below. Each column represents a processor mode, showing the available register resources.

Modes							
Privileged modes				Exception modes			
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt	
R0	R0	R0	R0	R0	R0	R0	
R1	R1	R1	R1	R1	R1	R1	
R2	R2	R2	R2	R2	R2	R2	
R3	R3	R3	R3	R3	R3	R3	
R4	R4	R4	R4	R4	R4	R4	
R5	R5	R5	R5	R5	R5	R5	
R6	R6	R6	R6	R6	R6	R6	
R7	R7	R7	R7	R7	R7	R7	
R8	R8	R8	R8	R8	R8	R8_fiq	
R9	R9	R9	R9	R9	R9	R9_fiq	
R10	R10	R10	R10	R10	R10	R10_fiq	
R11	R11	R11	R11	R11	R11	R11_fiq	
R12	R12	R12	R12	R12	R12	R12_fiq	
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq	
PC	PC	PC	PC	PC	PC	PC	
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Figure 2.1. ARM Register Organization [ARM05]

Program Status Registers include the Current Program Status Register (CPSR) and the Saved Program Status Register (SPSR). The CPSR is accessible from any of the processor modes, but the SPSR is only accessible from exception modes. The CPSR contains condition code flags, interrupt disable bits, current processor mode, as well as additional status and control information. The SPSR preserves the value of the CPSR whenever an exception occurs. Both the CPSR and SPSR have reserved bits, user-

writable bits, privileged bits, and execution state bits. Condition codes consist of Negative (N), Zero (Z), Carry (C), and oVerflow (V) [ARM05].

Exceptions occur when there is an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions interrupt normal execution flow, so processor state must be saved prior to executing the exception routine. ARM supports seven exception types as shown in Table 2.2 with the associated processor mode.

Table 2.2. ARM Exception Types [ARM05]

Exception type	Mode	VE ^a	Normal address	High vector address
Reset	Supervisor		0x00000000	0xFFFF0000
Undefined instructions	Undefined		0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor		0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort		0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0	0x00000018	0xFFFF0018
		1	IMPLEMENTATION DEFINED	
FIQ (fast interrupt)	FIQ	0	0x0000001C	0xFFFF001C
		1	IMPLEMENTATION DEFINED	

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.

ARM supports mixed endian data access—the address of a particular byte in memory will be the same regardless of whether it is being accessed through big endian or little endian means. Byte, halfword, and word accesses all return the same data regardless of endianness. This is accomplished through the use of byte invariance, which means that the address of a byte in memory is the same no matter what type of access is used. Double and multiple word accesses are treated as series of word accesses, so the same bytes are returned in these cases as well. Instruction fetches in ARM use little endian byte order and must be word-aligned.

ARM has unaligned word and halfword data access support. If enabled, the processor uses as many memory accesses as necessary to generate the required transfer of adjacent bytes transparently to the programmer.

ARM also supports comprehensive non-blocking shared-memory synchronization primitives that scale for multiple-processor system designs. This is an improvement over the read-locked-write operations that swap register contents with memory for shared memory synchronization. The two instructions that perform this synchronization are Load-Exclusive (LDREX) and Store-Exclusive (STREX). LDREX loads a register from memory, forces the executing processor to indicate an active inclusions access in the local monitor, and marks the physical address as exclusive access for the executing processor if the Shared memory attribute is present for this address. STREX performs a conditional store to memory, only if the executing processor has exclusive access to the memory addressed.

The Jazelle Extension accelerates bytecode execution of Java Virtual Machines (JVMs) [ARM05]. JVMs can be written to automatically take advantage of accelerated opcode execution if available, but the bytecode will still execute even if the extension is not present. The Jazelle Extension expects general-purpose registers and other resources to conform to a particular calling convention when the Jazelle state is entered and exited. The J bit from the processor status registers in conjunction with the T bit determines the execution state of the processor.

Finally, saturated integer arithmetic is supported in ARM. Saturated arithmetic modifies the way normal integer arithmetic behaves by allowing arithmetic operations that exceed the bounds of the 32-bit registers. The result of a saturated arithmetic

operation represents the closest possible number to the correct mathematical result. If the correct result is too great to represent in 32 bits and overflows the upper end of the representable range, the result is set to $+2^{31}-1$. If the correct result is too small to represent in 32 bits and overflows the lower end of the representable range, the result is set to $-2^{31}-1$. This modification is useful for many Digital Signal Processing (DSP) applications. These applications do not react well to an abrupt change of sign, which would be the result on an arithmetic operation overflow.

2.1.2 Instruction Set

ARM instructions must adhere to the specific encoding pattern shown in Figure 2.2. Any other pattern of bits is considered UNPREDICTABLE or UNDEFINED. Most ARM instructions will act as a NOP unless the N, Z, C, and V flags in the CPSR satisfy the condition specified in the condition code field of the instruction. There are only a few instructions that execute unconditionally. These instructions have been introduced in ARMv5 and later.

Branch instructions allow conditional branches either forward or backward in the program. These branches can be up to 32MB and are executed by a specific instruction or by writing a value to the program counter (PC) register. Additional functionality is introduced with the Branch with Link (BL), Branch and Exchange (BX), Branch with Link and Exchange (BLX), and Branch and Exchange Jazelle (BXJ) instructions.

														31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing immediate shift	cond [1]	0	0	0	opcode			S	Rn		Rd	shift amount			0	shift		0	Rm																										
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x													
Data processing register shift [2]	cond [1]	0	0	0	opcode			S	Rn		Rd	Rs		0	shift	1	Rm																												
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x															
Multiples: See Figure A3-3 Extra load/stores: See Figure A3-5	cond [1]	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x															
Data processing immediate [2]	cond [1]	0	0	1	opcode			S	Rn		Rd	rotate		immediate																															
Undefined instruction	cond [1]	0	0	1	1	0	x	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x															
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask			SBO	rotate		immediate																													
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn		Rd	immediate																																
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn		Rd	shift amount		0	shift	0	Rm																											
Media instructions [4]: See Figure A3-2	cond [1]	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x																
Architecturally undefined	cond [1]	0	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x																
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn		register list																																	
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset																																							
Coprocessor load/store and double register transfers	cond [3]	1	1	0	P	U	N	W	L	Rn		CRd	cp_num		8-bit offset																														
Coprocessor data processing	cond [3]	1	1	1	0	opcode1			CRn	CRd		cp_num	opcode2		0	CRm																													
Coprocessor register transfers	cond [3]	1	1	1	0	opcode1			L	CRn		Rd	cp_num		opcode2	1	CRm																												
Software interrupt	cond [1]	1	1	1	1	swi number																																							
Unconditional instructions: See Figure A3-6	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x																

1. The cond field is not allowed to be 1111 in this line. Other lines deal with the cases where bits[31:28] of the instruction are 1111.
2. If the opcode field is of the form 10xx and the S field is 0, one of the following lines applies instead.
3. If the cond field is 1111, this instruction is UNPREDICTABLE prior to ARMv5.
4. The architecturally Undefined instruction uses a small number of these instruction encodings.

Figure 2.2. ARM Instruction Encoding Pattern [ARM05]

BL preserves the address of the instruction after the branch. BX copies the contents of a general-purpose register to the PC and shifts the processor to Thumb state if bit [0] of this transferred value is 1. BLX behaves like BX, but writes the address of the next instruction into the LR and shifts to Thumb state. BXJ also behaves like BX, but enters Jazelle state if it is available and enabled. These instructions implement subroutine behavior if the programmer desires to use them as such.

ARM provides 16 different data-processing instructions to perform logical operations, basic arithmetic operations, tests, comparisons, moves, and bit clears. Most of these instructions require two source operands. Some store results to a register and update condition flags and others simply update condition flags for jump and branch operations (conditionals and loops). One of the two source operands must always be a register and the other may be a register or an immediate value, depending on the specific instruction.

ARM can perform multiplication on several different classes of instructions. Normal multiplication takes two 32-bit inputs and returns a 32-bit output. Long multiplication takes two 32-bit inputs and returns a 64-bit result. Halfword multiplication takes two signed 16-bit inputs and returns a 32-bit result. Word, halfword multiplication produces a top 32-bit result. Most significant word multiplication takes two 32-bit inputs and returns a top 32-bit result. Dual halfword multiplication produces a 32-bit result from two 16-bit inputs.

ARM's normal data-processing and multiply instructions are complemented by a set of parallel addition and subtraction instructions. There are six distinct basic instructions, each of which has six variants, for a total of 36 possible instructions. The basic instructions exchange or manipulate the data sources while the variants incorporate signed/unsigned arithmetic modulo 2^8 or 2^{16} , signed/unsigned saturating arithmetic, and signed/unsigned arithmetic with halved results. Similarly, extend instructions come with six basic instructions that unpack data by sign or zero-extending bytes to words/halfwords and halfwords to words. There are sign extension and zero extension variants for each of these six basic extend instructions.

Two instructions move contents of a PSR to or from a general-purpose register. There are also several instructions that write directly to specific bits or groups of bits within the CSPR. These instructions can set a condition code flag to a known value, to enable or disable interrupts, to change processor mode, to change the endianness of load/store operations, and change the processor state.

The basic load and store instructions in the ARM architecture come in two broad types. The first loads or stores a 32-bit word or an 8-bit unsigned byte. The second loads or stores a 16-bit unsigned halfword, load and signs a 16-bit halfword or an 8-bit byte, or loads or stores a pair of 32-bit words. Addressing modes for both types are formed using the base register and an offset. The base register is always a general-purpose register and the offset is an immediate value, a register, or a scaled register. This combination of base register and offset forms the memory address in one of three ways: offset, pre-indexed, or post-indexed. Multiple load and store instructions are similar in format and intent except they operate on a subset of the general-purpose registers rather than one at a time.

The Swap (SWP) or Swap Byte (SWPB) instructions operate on semaphores. Both instructions have a single addressing mode and are used for process synchronization. Memory semaphores can be loaded and altered without interruption because the load and store operations are atomic.

Processor exceptions occur via a Software Interrupt (SWI) instruction or a Breakpoint (BKPT) instruction. User mode can make calls to privileged OS code by using the SWI instruction. The BKPT instruction causes a Prefetch Abort exception to occur, which is handled by a previously installed debug monitor program. This is

sometimes referred to as a software breakpoint. The ARM processor ignores the immediate fields in both of these instructions.

Coprocessor instructions in the instruction set provide communication with coprocessors. The three types include a coprocessor data processing operation, register transfer to and from coprocessor registers, and address generation for the coprocessor Load and Store instructions [ARM05]. Coprocessors are distinguished by a 4-bit field in the instruction.

2.1.3 Addressing Modes

The first addressing mode used with ARM instructions is called “Data-processing operands.” This mode has 11 formats to calculate the shifter_operand portion of the data-processing instruction. This shifter_operand portion could be an immediate, a register, or the result of one of many shift or rotate operations on a register. Each variation of the 11 formats has its own specific syntax and operation flow.

The second addressing mode is “Load and Store Word or Unsigned Byte.” The mode has nine formats to calculate the address for the respective load or store instruction. The addressing_mode portion of the load or store instruction could be an immediate offset/index, a register offset/index, or a scaled register offset/index. For an immediate offset, the address is calculated by adding or subtracting the immediate value to or from the value in the base register. For a register offset, the mode calculates an address using the values in the index register and the base register. For scaled register offset, the mode calculates the address using the shifted or rotated value in the index register and the base register.

The third addressing mode is “Miscellaneous Loads and Stores” with six formats. The addressing_mode portion of the load or store instruction could be an immediate offset, register offset, immediate pre-indexed, register pre-indexed, immediate post-indexed, or register post-indexed. The addressing_mode portion of the instruction is calculated in the same way as the second addressing mode, above.

The fourth addressing mode is “Load and Store Multiple.” These instructions work the same as those above in the third addressing mode except that they operate on a subset of the general-purpose registers rather than a single register. The addressing_mode can be increment after, increment before, decrement after, or decrement before. For increment after, the start_address is equal to the base register value and increments by four for each subsequent address. For increment before, the start_address is equal to the base register value plus four and increments by four for each subsequent address. For decrement after, the start_address is equal to the base register value minus four times the number of registers specified in the encoding, plus 4 and increments by four for each subsequent address. For decrement before, the start_address is equal to the base register value minus four times the number of registers specified in the encoding and increments by four for each subsequent address.

The final addressing mode is “Load and Store Coprocessor.” This mode has four options to calculate the address of a respective load or store instruction. The addressing_mode could be an immediate offset, immediate pre-indexed, immediate post-indexed, or unindexed. All four options produce a sequence of consecutive addresses. For immediate offset, the mode adds or subtracts four times the immediate offset value to or from the base register value to get the first address and increments by four for

subsequent addresses until signaled by the coprocessor to stop (no more than 16 words). For immediate pre-indexed, the mode adds or subtracts four times the immediate offset value to or from the base register value and increments by four for subsequent addresses until signaled by the coprocessor to stop. The difference is that the first address is written back to the base register only when the condition code status matches the condition specified in the instruction. For immediate post-indexed, the first address is the base register value and the mode increments by four for subsequent addresses until signaled by the coprocessor to stop. The base register value is updated during the process whenever the condition code status matches the condition specified in the instruction. For unindexed, the first address is the base register value and the mode increments by four to calculate subsequent addressed until signaled by the coprocessor to stop.

2.1.4 Memory and System Architectures

Memory behavior in ARM is classified by type: strongly ordered, device, and normal. Each of these types can be further distinguished by access mechanisms and cacheable and shared attributes. Coprocessor 15 (CP15) is the primary control mechanism for virtual memory systems as well as identification, configuration, and control of other memory configurations and system features.

The type, size, access speed, and architecture of memory are all important parts of the decision process to achieve certain overall system performance and cost goals. A memory hierarchy is formed when different types of memory are included in a system design. The memory is typically layered where layers with higher numbers are further

from the core and have increased access times. ARM provides caches and I/O at each layer. Higher layers have a larger size but also increased latency.

The L1 cache supports multiple virtual address aliases to a specific memory location. CP15 controls the size, associativity, and organization parameters of the cache within the subsystem. Entries in the L1 cache do not need to be invalidated for different virtual to physical mappings. This reduces the requirement for cache clean on a context switch, which helps software perform more efficiently. Aliases to the same physical address may exist in memory regions that are described in the page tables as being cacheable [ARM05]. The L2 cache can be either tightly coupled to the core or implemented as memory mapped peripherals on the system bus. Additional levels of cache may be used, but are not required.

Tightly Coupled Memory (TCM) is a physically addressed area of memory that makes up part of the Level 1 memory subsystem (along with the L1 cache). This area provides low latency memory without the unpredictability of caches. This memory is ideal for storing critical routines, for use as scratchpad data, for data types whose locality properties are not well suited to caching, and for critical data structures such as interrupt stacks [ARM05].

Resets, interrupts, and imprecise aborts are typically asynchronous events, as opposed to the synchronous events tied to many exceptions. Resets are the only non-maskable event contained within the ARM architecture. Interrupts have three different levels: fast interrupt request, non-maskable fast interrupt request, and normal interrupt request. Whatever causes the interrupt must be deasserted prior to re-enabling of the interrupts.

2.1.5 Vector Floating-Point Architecture

The vector floating-point architecture (VFP) is a coprocessor extension to the ARM architecture [ARM05]. It adds single-precision and double-precision floating-point arithmetic to the system. To completely implement the VFP, the architecture must include support code which provides features not supplied by the hardware.

The VFP comes with 32 general-purpose registers, and a full set of instructions for loading, storing, transferring, adding, subtracting, multiplying, dividing, square-rooting, copying, converting, and comparing values in these registers. VPF also supports floating-point exceptions for invalid operations, division by zero, overflow, underflow, and inexact.

VFPs can be implemented with or without a hardware component. Software-only implementations (VFP emulators) use ARM routines to emulate all floating-point arithmetic. These implementations can be more efficiently accomplished through the direct use of software floating-point libraries, and hence have not been developed. Hardware implementations use the hardware to handle common cases and use support code only when the hardware cannot handle a case. This approach optimizes the performance of the architecture.

2.2 Introduction to Google Android Operating System

Release of the Google Android OS opened numerous opportunities for coders and application developers to write programs and make modifications to customize almost any portion of a mobile device [FOG09, Dim08, Has08, BurE09]. This open platform supports customized legitimate applications, but also opens the door for a significant

increase in malicious content [RML09]. The Android OS is primarily for use on mobile devices, mainly cellular phones [Mur09]. Organizations target customers who own mobile devices and companies who want their core applications built on a platform supported by the Open Handset Alliance (OHA) [HK09], so it is important that security professionals develop an understanding of Android OS to mitigate risks and vulnerabilities.

This section presents background on Android, followed by specific advantages and known implementations of Android. The information contained in this section provides a better understanding of the impact Google Android OS has had on the mobile device community.

2.2.1 Operating System Background

Google was among the first in the mobile OS community to open mobile OS's by developing the Android Platform, supporting standards and publishing APIs which encouraged widespread, low-cost development of mobile applications. In September 2008, T-Mobile released the first smartphone based on the Android Platform as well as a Software Development Kit (SDK) [UTG08]. In October, the source code was made available under Apache's open source license.

Key architectural goals of the Android Platform allow applications to interact with one another and to reuse components. The platform incorporates a Linux-based operating system stack for managing devices, memory, and processes and has libraries related to telephony, video, graphics, and User Interface (UI) programming [HK09].

The Android architecture consists of five distinct layers on the system stack: the Acorn RISC Machine (ARM) Linux core, the libraries, the Dalvik run-time byte-code interpreter, the application framework, and the applications [JTD09]. The platform is not a single piece of hardware or software, but a complete end-to-end software framework configurable to work on a variety of hardware implementations. It includes everything from the bootloader to the applications. Figure 2.3 shows a graphical representation of the application stack.

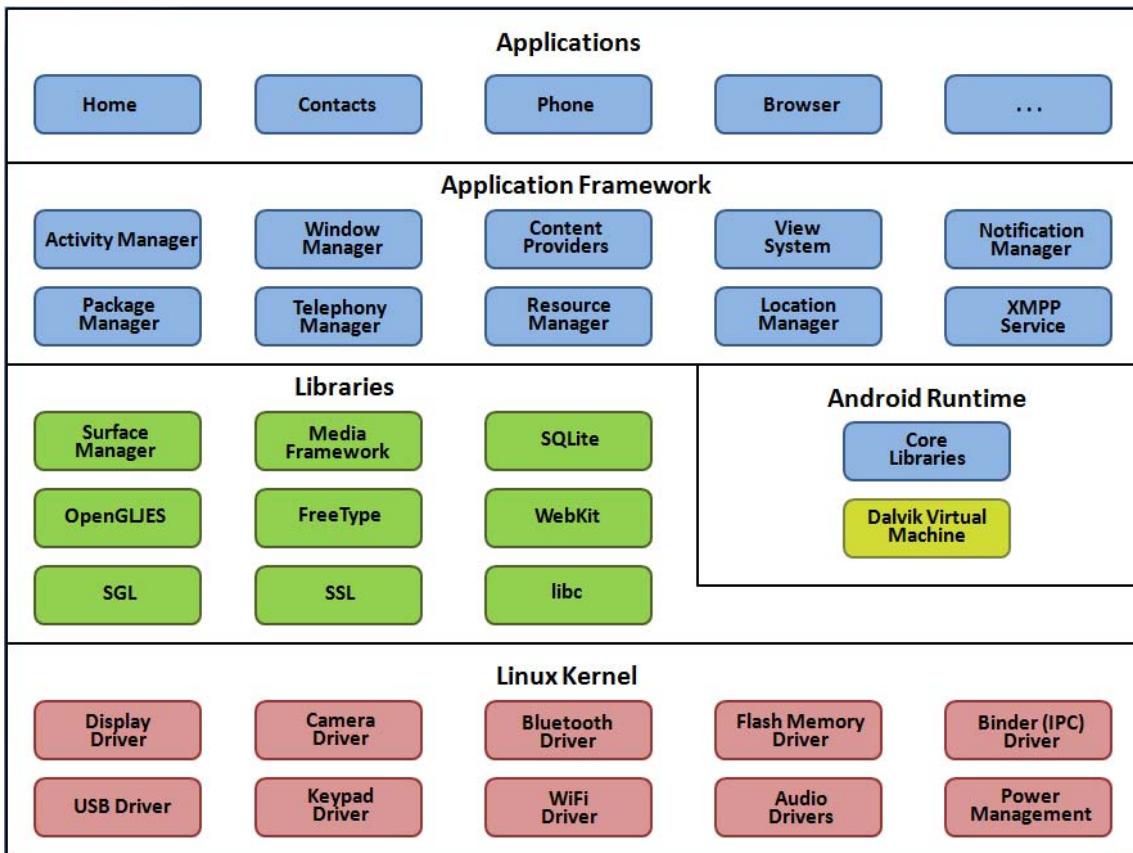


Figure 2.3. Overview of the Android Application Stack [SDT08]

2.2.2 Advantages Offered by Android

The Android Platform offers a variety of advantages not currently available in other mobile operating systems. Google opened the Android market, allowing

application developers to publish applications without any restrictions [DCI09]. Additionally, being an open platform encourages device and service provider-independency. Consumers are not tied to a specific device or cellular-service company to use Google Android.

Android provides fully-developed features to exploit cloud-computing resources and supports a relational database on the handset [HK09]. It supports 2D and 3D graphics as well as various media file formats, allowing developers to create media common applications [DCI09].

The Dalvik VM significantly enhanced the power management system of the Android Platform. This custom VM takes generated Java class files and combines them into its own native executable format. Since it reuses duplicate information across various class files, space requirements are half what the JVM .jar file requires [HK09]. Google also fine-tunes the garbage collection, omits the just-in-time (JIT) compiler, and uses registers instead of the stack for generation of assembly code. These enhancements significantly reduce the power requirements of the system, making the Android Platform suitable for mobile device use.

Finally, Android application developers can develop applications for any platform [JMH08] and applications can run in parallel when loaded on the device. This allows processes running in the background to send alerts and notifications to the user.

2.2.3 Known Implementations of Operating System

The Open Handset Alliance (OHA) is a confederation of 50 Telecoms, mobile hardware, and software companies. Headed by Google, the OHA backed Android as one

of its first open platform operating systems. The Android Platform has been released on numerous cellular phones across a wide variety of service providers.

Good Technology uses Android devices to connect to their corporate enterprise so employees can access company resources via a secure container in the client which separates protected enterprise data from personal data and applications stored on the mobile device. The container also enables the IT department to enforce security policies, wipe enterprise data, and have government-grade data encryption [GT09].

The FrauVent application improves the physical security of sensitive information utilized during financial transactions [PGT09]. FrauVent incorporates a multi-modal protocol that gives users information about a pending questionable transaction in a way that provides a suitable context for approving or rejecting such exchanges. The goal is to establish the legitimacy of the transaction. FrauVent uses the GPS and Mapping capabilities resident on an Android device. For example, when questionable charges are applied against a user's bank account, the financial institution immediately sends a message to the user's phone requesting location and purchase information. The user has the opportunity to follow reactive protocol and approve or flag the transaction. Users can also follow proactive protocol and send location verification to their financial institute prior to making transactions. Proactive action prevents account lockouts and fraud flags when transactions are made at odd hours or in varied locations. This solution reduces the costs of fraud without requiring financial institutions to significantly change their extensively deployed end systems.

Android supports memory streaming, making it suitable for Voice over Internet Protocol (VoIP) and there are proposals to incorporate software in Android devices to

secure VoIP [YA09]. Additionally, the automotive industry may incorporate the Android Platform into In-Vehicle Infotainment systems [MTV09]. The open platform encourages reuse between models as well as between manufacturers. The capabilities of Android provide an interface similar to Personal Digital Assistant (PDA) and cell phone interfaces consumers have come to expect.

IMS-Learning Design (IMS-LD) learning activity-based implementations rely on client-server architectures which are problematic for resource-limited mobile devices without reliable Internet access [ZNA09]. Google Android implements a subset of the IMS-LD design specification and uses SMS messages for synchronization thereby providing a correlated learning environment for system users.

Mobile Social Networks (MSNs) are also making use of Android [LC09]. The information stored on devices is often shared and transferred between members of these networks. MSN applications collect and store data on the device as well as information pertaining to any social network contacts or “friends.”

2.2.4 Summary

This section introduces the Google Android OS, examines the background of the development as well as some of the features the platform offers over competing mobile operating systems, and outlines several uses of the Android Platform in various capacities. It is most prevalent in mobile devices, but is also starting to be used in corporate networks, the automotive industry, and the banking industry to secure financial transactions. With the widespread use of this platform, it is imperative that security

mechanisms be thoroughly reviewed and improved to protect data throughout various implementations.

2.3 Examination of Android Protection Mechanisms

This section reviews components within Android and briefly describes the associated interactions. The built-in security features of the OS are closely examined. Three current implementations of security measures for Android are also reviewed. The strengths and weaknesses of each method are discussed. Finally, an improved Android security protection mechanism is proposed.

2.3.1 Component Interactions

Android defines four component types: Activity, Service, Content provider, and Broadcast receiver [EOM09]. Activity components define an application’s user interface. Only one activity has keyboard and processing focus at a time, all others are suspended. Services do background processing thereby enabling activities to continue after the user interface disappears. Content providers store and share data using a relational database. Each one has an associated authority describing the content it contains. Broadcast receivers act as mailboxes for messages from other applications.

2.3.2 Built-in Security Features

The validity of on-board security features are a key interest area for consumers [Tho09]. Natively, Android provides protection through permissions as well as isolation and signatures. Permissions ensure that explicit access is granted by an application for other applications to access data and functionality. These permissions cannot be set at run-time, but rather must be set at install time via a “manifest” which contains the

permissions enforced and requested by each application [BurJ08]. When a package installer is installing an application, it sets all of these permissions in the manifest via dialog with the user. This is flawed in that it is not actually known whether applications will use the permissions and thereby gain trust legitimately or not [Phy10].

The isolation and signatures protection native in Android are implemented by running applications in their own Virtual Machine (VM) and as a Linux process. Each application is assigned a unique Linux user-id (UID) so its files are not visible to other applications. This allows Android to limit the damage of any programming flaws. If signatures allow UIDs to be shared, files can become visible to other applications [BurJ09].

Currently, Android does not support hardware-based security features for application developers, although most Android phones are equipped with the required hardware modules [SAH09]. Android does, however, provide an additional protection in the form of signatures. Any Android application must be signed with a certificate whose private key is held by the developer. This certificate does not need to be signed by a certificate authority; it is used only to establish trust between applications by the same developer [Cha09, SFK10]. This signature does not provide complete protection, but adds an additional layer of security to the overall system.

A team from Kokusai Denshin Denwa Institute (KDDI) R&D Laboratories formally analyzed the permission-based security model of Android, showing that after specifying system elements, the specified system preserves the desired security properties [SKF09]. This analysis was based on certain specific states, but does not translate to all states an Android device could enter.

2.3.3 Current State of Protection Mechanisms

Several protection mechanisms for the Android Platform have been developed. This section examines three of them and explores the strengths and weaknesses of each. SCanDroid provides users with a better context for making security-relevant decisions when installing applications. Saint (Secure Application INTeraction) governs install-time permission assignment and their use at runtime. Finally, static analysis of executables uses collaboration to accomplish malware detection.

2.3.3.1 SScanDroid

SScanDroid (Security Certifier for Android) reasons about the security of Android applications [FCF10]. It statically analyzes data flows through applications and makes security-relevant decisions automatically. This provides the user context to make an informed security decision when installing a new application. An Android application can allow other applications to share its data and functionality, but the accesses must be carefully controlled.

SScanDroid relies on Android-provided access controls and on underlying abstract semantics of Android applications to track data flow through and across components, as shown in Figure 2.4. The implementation consists of seven modules as well as Watson Libraries for Analysis (WALA) interface: a bytecode loader, string/data analysis, an inflow filter, flow analysis, an outflow filter, a manifest loader, and a checker. The bytecode loader sends application bytecode through String/Data Analysis and Flow Analysis, resulting in a Flows for Application consisting of data flow maps and graphs. The Checker compares this Flows for Application to output from the Manifest Loader

determining Constraints for the application. If data flow is not consistent with security permissions specified by the manifest, the user is informed of potential danger prior to application installation. Not all data flow can be statically analyzed, so some Constraints may be conditional, requiring Additional Information for Further Analysis.

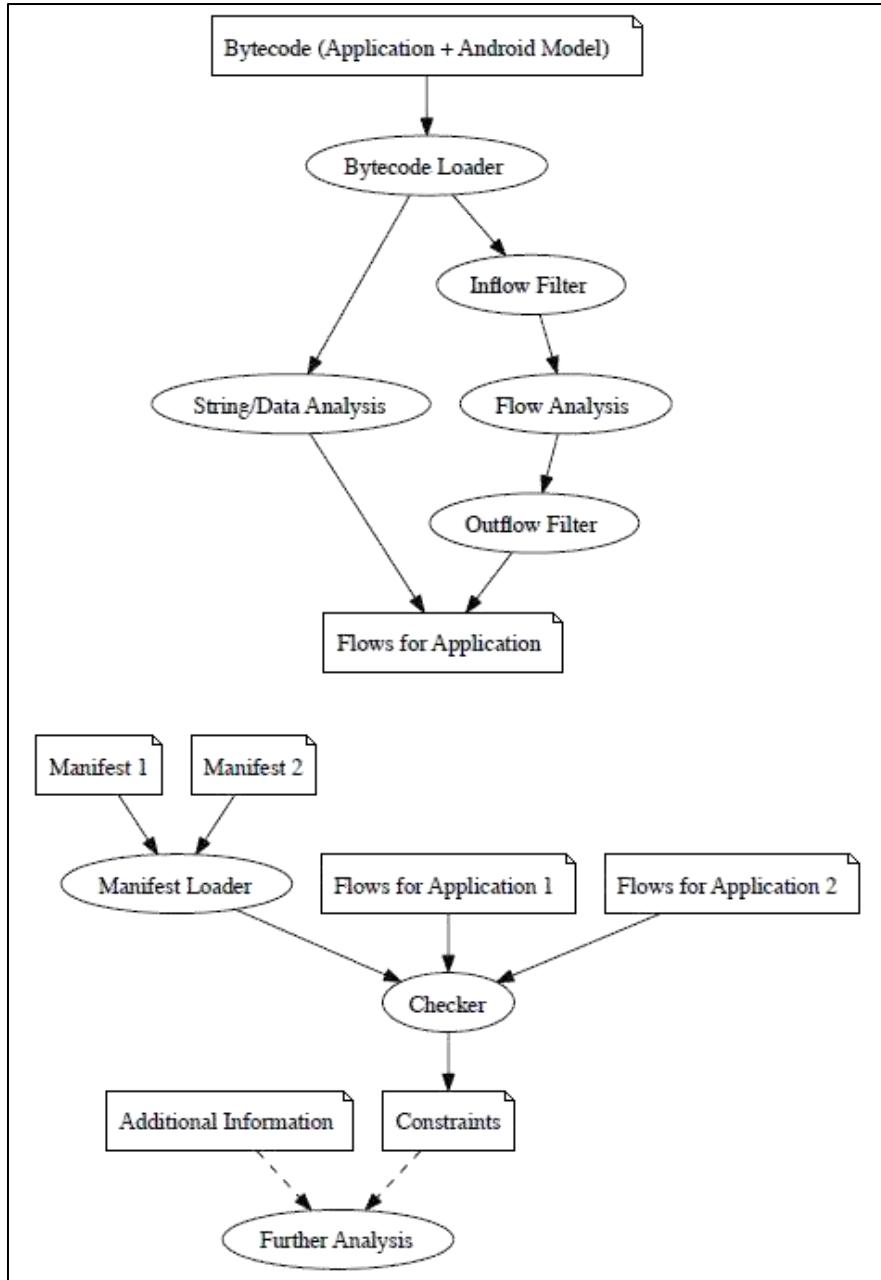


Figure 2.4. SScanDroid Architecture of Analysis [FCF10]

Android applications all have components of type activity, service, broadcastReceiver, and contentProvider [BurJ08]. Components extend one of the base classes and override the methods in that class. Each of the methods is considered an entry point and SCanDroid modularly analyzes those entry points. SCanDroid treats component classes and methods as idealized, primitive constructs (as opposed to modeling general classes and methods). It considers permissions as the only mechanism to control cross-component interactions. This ensures data cannot flow from one store to another by preserving a well-typed environment. Stores are generalizations of content providers, databases, files, and other data containers. Therefore, a value from m can flow to store n only if readers of n can already read from m and writers of m can already write to n.

This system requires the Java source code of the compiled JVML bytecode of applications for analysis. Depending on the source of applications, this bytecode may be difficult to obtain. Additionally, this system still allows the security decision to be made by the end user. End users tend to be more focused on convenience and availability than on security issues.

2.3.3.2 Secure Application INTeraction (Saint)

Saint addresses the limited ability of applications to control who can access their interfaces as well as compensates for the rudimentary facilities that control how their interfaces are used by other applications. Finally, Saint enhances the limited means applications have of selecting which application's interfaces they use [OME09]. The

improved infrastructure provides applications with installation-time policies to regulate the assignment of permissions that protect their interfaces.

Saint uses an enhanced installer for applications to regulate application-defined permissions. This goes well beyond the Android model of only allowing/disallowing permission assignments based on application-independent rules. Now applications can exert control over the assignment of permissions declared through an explicit policy [OME09]. Saint enforces runtime policies of two types: access policies for identifying caller security requirements and expose policies for identifying callee security requirements. Saint optionally can allow or disallow the user to override system/application policies.

The Saint Installer and Saint Mediator are the key components of the Saint architecture, along with an AppPolicy Provider, FrameworkPolicyManager, and Condition Extensibility. The Installer and Mediator enforce additional permission granting policies and mediate interprocess communication to ensure interaction policies specified by both the caller and callee applications are enforced. This greatly enhances the native Android permission security, but still allows users to determine which applications to install and run.

2.3.3.3 Static Analysis of Executables

Schmidt et al [SBS09] statically analyze executables on Android for collaborative malware detection. They extract function calls from the Android environment using the command readelf and compare this attribute set with malware executables, using PART, Prism, and Nearest Neighbor Algorithms for classification.

The PART classifier scans the decision tree learner and extracts decision rules. The Prism classifier uses pure rules to cover the entire attribute set through rule induction. The Nearest Neighbor Algorithm maps each result to a state space of {malicious, normal}, calculates distance to a subset, and determines if this result falls within a specified uncertainty level.

The binary determination between malicious and normal executables is transformed into a certainty value falling in the range [0, 1]. A value of 0 indicates normal and a value of 1 indicates malicious. Values in between indicate level of maliciousness. Taking desired false-positive rate into consideration, a threshold is set for distinguishing between normal and malicious content. Depending on the results, analysis can be performed on-device, sent out for collaboration between mobile devices, or sent to a remote mobile server for further inspection. The executable is then classified as benign if it falls below the threshold; otherwise, it is classified as malicious and not allowed to run.

The weakness in this method is it requires a device to trust other surrounding devices if acceptable results are not obtained on the device itself. There is no guarantee that a neighboring device has not been compromised and will provide misleading analysis results. Additionally, since the system compares known malware function calls to function calls of legitimate executables, it is a form of signature-based detection. If a malware developer uses function calls that closely follow legitimate executables already on the Android device, the malware stands a good chance of being classified as benign.

2.3.4 Proposal for Improved Security Measures

On March 9, 2010, England’s The Register reported an instance of an embedded malware on an HTC Android phone that attempted to steal information from connected personal computers (PCs) when the device synchronized [Gir10]. The malware itself was resident on a Secure Digital (SD) memory card mounted in the device. This type of malicious behavior will become more prevalent on mobile devices as mobile use of data networks increases. None of the current security implementations discussed in this chapter would have prevented the attack described.

In fact, malicious code can be executed on mobile devices despite all precautions the end user may take to avoid unintentionally allowing programs to access or modify data outside the parameters of set permissions. A protection mechanism for PCs called SecureQEMU [Kim09] requires all legitimate code on the machine be signed at the page level. Hashes for each page are protected in the system kernel. Only code executing within a signed page is allowed to execute on the machine. Code attempting to execute on the machine is checked against the page hashes stored in the kernel and if there is not a match, none of the code on that page is allowed to execute. This system requires a known good state from which to initialize the protection and be provided trusted hashes. To date, no similar protection mechanism has been implemented on a mobile OS.

2.3.5 Summary

This section presents an overview of protection mechanisms native to the Android Platform. It starts with a brief description of system components and then explores three aspects of built-in security within the OS. It presents three alternative protection

mechanisms developed for the Android, examining the strategy behind each method. Finally, it concludes none of these protection mechanisms are sufficient to protect a device from execution of malicious code. A brief overview of a new mobile security solution is provided and is examined in depth in the following chapters.

III. Methodology

3.1 Background

Google’s Android operating system (OS) is an open platform, allowing programmers to modify and customize the content and operational environment of mobile devices. Malicious code can be executed on mobile devices despite all precautions an end user may take to avoid unintentionally allowing programs to access or modify data. This research takes protection mechanisms originally developed for personal computers (PCs) and moves them to the Android environment. Legitimate code is signed at the application package level for all programs on the device in a known good state. Hashes for each package are stored in the system kernel and only code from a signed package is allowed to execute on the machine. Code attempting to execute on the machine is checked against the package hash stored in the kernel and if there is no match, code in that package is not allowed to execute. This system requires a known good state from which to initialize the protection and be provided trusted hashes. As the world’s computing environment becomes more and more mobile, it is crucial that the same security precautions employed on PCs are transitioned into the mobile environment.

3.2 Problem Definition

This section describes the specific goals of the research along with a hypothesis of the expected results. The approach describes how the hypothesis is tested against the research goals.

3.2.1 Goals and Hypothesis

The explosion of laptop and handheld devices around the world has significantly increased the importance of the mobile computing environment. Cell phones are no longer simply a means of making person-to-person calls, they now store and transfer data, play music, check and send e-mail, browse the Internet, receive GPS navigation, and more. As a result, like PCs, mobile devices are increasingly the target of malicious attacks. The goal of this research is to provide a robust protection mechanism for mobile OSs. The research uses as a baseline a mechanism implemented on a PC and determines if it can provide the same level of protection on a mobile platform. Specifically, the research determines the effectiveness of implementing system protection within the kernel of the OS itself. The effectiveness of the new protection is compared to the native protection offered by the OS.

The data collected during testing is examined to analyze the various protection levels offered and system overhead. The hypothesis for the research is that the new protection method will provide significant improvement in mobile system security without requiring substantial overhead. It is expected an end-user will notice little to no difference in system performance once the new protection mechanism is in place.

3.2.2 Approach

Many mobile protection programs run on a mobile device as just another user program. A better approach is to embed the protection mechanism within the kernel itself. The kernel is modified such that it recognizes legitimate code and programs

requesting to execute on the device. Only recognized programs are allowed to execute.

This is known as a white-list.

To determine if the new protection mechanism performs better than the native protection of the system, unapproved application packages are submitted to the mobile device. The success rate of the new protection mechanism is compared to the native protection success rate.

To determine whether the improved protection system requires substantially more overhead than the existing protection system, the load time for various programs with and without the new protection mechanism enabled is also analyzed.

3.3 System Boundaries

The System Under Test (SUT) for this research is the Android Protection System (APS). APS includes the Android mobile device, the Android OS, a modified kernel, and various default applications on the mobile platform. Approved and unapproved applications provide input to the system but are not part of the system itself. This research focuses on Google's Android platform, built on the ARM processor architecture. No additional OS's are considered. The SUT does not include any Android applications that may provide system protection, only the native protection is in place. The Android mobile device communicates with an external server to receive input and updated system content. These interactions are guided by the end-user. The workload on the system consists of an end-user performing standard mobile device functions such as placing phone calls, browsing the Internet, sending text messages, running applications, and

listening to music. APS prevents unapproved content from executing on the Android device.

The Component Under Test (CUT) is the modified kernel within the Android OS. Figure 3.1 shows the system complete with inputs, outputs, and internal components.

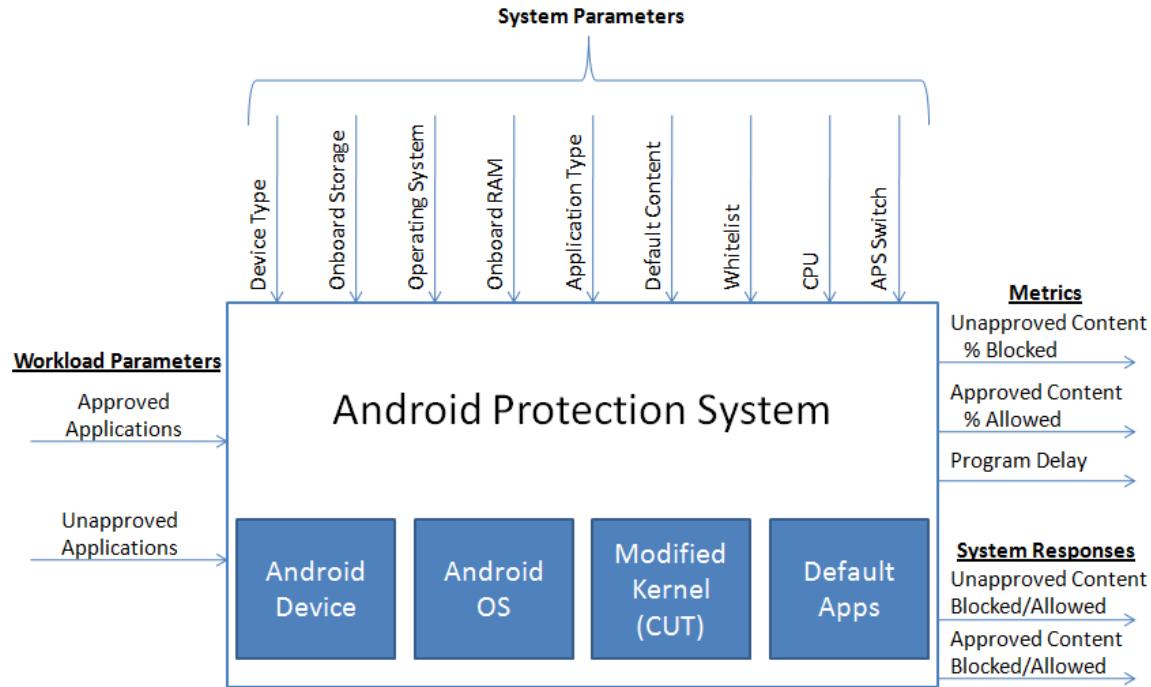


Figure 3.1. Android Protection System

3.4 System Services

The service APS provides is protection from the execution of unapproved content on the mobile device. This allows for normal use of all approved programs and applications on the device while disallowing all others. Any unapproved program attempting to execute is considered malicious and the protection service prevents execution. The protection service does not interfere with the execution of approved programs and applications.

The protection service has two primary outcomes: unapproved content is successfully blocked or it is allowed. These two outcomes are each paired with a secondary outcome: approved content is successfully executed or approved content is blocked. The primary focus of the system is to successfully prevent unapproved content from executing on the mobile device. However, a mobile device is rendered unusable if approved content is also unable to execute. It is critical that the protection service succeed in both the primary and the secondary outcomes.

3.5 Workload

The workload for the system consists of programs within the system. The workload includes both approved and unapproved applications. The end-user submits this workload to the system by running programs and applications on the Android device.

The number and size of applications sent to the system are workload parameters. These parameters vary from small to medium to large to extra-large levels and are discussed further in Section 3.8. When varied, these requests can affect the performance of the system. It is important to vary the workload to determine how the system performs under different loads. Varying the workload parameters ensures that the system blocks all unapproved content without noticeable performance degradation.

3.6 Performance Metrics

The first metric to evaluate the performance of the APS is the percentage of unapproved content blocked. This metric directly reflects the success of the system in preventing the execution of unapproved programs or applications on the Android device.

To achieve success, APS needs to achieve 100% for this metric; all unapproved content is blocked.

The second metric is the percentage of approved content allowed. This metric also reflects how well the system allows all approved programs and applications to execute on the Android device. To succeed, APS needs to achieve 100% for this metric as well; all approved content is allowed.

The final metric is program delay. This metric compares the measured time for program and application loading without the protection mechanism in place to the measured time for loading with the protection mechanism in place. APS adds overhead because it verifies all execution requests prior to program execution. This delay manifests itself in the time it takes for the program or application to load. Once the program or application is executing, system overhead should be the same whether or not the protection mechanism is in place. The measurement of this metric starts when the system receives a program request and stops when the program completes initialization and is ready for user interaction. Therefore, this metric measures the load time for execution requests.

3.7 System Parameters

The parameters listed below affect performance of the protection mechanism.

- Device Type – The device type specifies the particular mobile platform being tested. This research uses an Android Developer Phone 2 (ADP2).
- Operating System (OS) – The OS changes how the protection mechanism is implemented while the hardware architecture changes based on the OS.

Modifications to the kernel or hooks into system calls are treated differently depending on the OS. The OS used in this research is Android OS 1.5.

- Onboard RAM – The RAM available determines how quickly the system will be able to process requests. More RAM should result in better performance. The ADP2 has 192MB of onboard RAM.
- Onboard Storage Space – The storage space available in the system determines how many programs and applications can be loaded on the device at a time. Higher capacity means more executable code can be loaded. The ADP2 has 512MB of Flash memory (ROM) and a 1GB microSD card.
- Application Type – The type of applications used in the test changes the way the protection mechanism behaves. Content submitted to the system includes Android application files.
- Default Content – The programs and applications installed by default on the device affect the performance of the protection mechanism. A larger collection of default content translates to more executable code that must be signed initially and more hashes that must be checked for verification purposes each time an execution service request is received. Default applications on the ADP2 include: Alarm Clock, Browser, Calculator, Calendar, Camcorder, Camera, Contacts, Dev Tools, Dialer, Email, Gallery, Messaging, Music, Settings, Spare Parts, and Voice Dialer.
- White-list – APS relies on a collection of hash digests of each approved application package in the system. The location and size of this white-list affects the efficiency of the protection mechanism.

- Central Processing Unit (CPU) – The CPU of the system device determines how fast the system can execute instructions and therefore has an effect on the efficiency of the protection mechanism. The ADP2 has a MSM7200A 528MHz processor.
- APS Control – APS consists of a modified Android kernel, complete with an application white-list and custom security functions. This protection mechanism is either enabled on the device or is absent. Enabling APS affects system performance.

3.8 Factors

The factors listed below are used in this research at the corresponding levels.

Table 3.1 displays a table of all factors and levels.

Table 3.1. Experimental Factors

Factors	Levels	
Application Type	Unapproved Content	Malicious Application
		Non-malicious Application
Approved Content		
Workload		Small
		Medium
		Large
		Extra-Large
APS Switch		Enabled On
		Enabled Off

- Application Type
 - Unapproved Content – This application is not on the approved list for the system and can be one of two types: malicious or non-malicious application.

- Malicious Application – This application uses a virus or malware to attack the system. The malicious application represents an unauthorized user trying to gain access to the system.
- Non-malicious Application – This program or application is not malicious but is also not on the approved list for the system. Non-malicious applications represent legitimate users trying to download and install programs or applications not approved for use on the device.
 - Approved Content – This application is on the system by default or included in the list of programs allowed to execute on the device. All executable code in this level is signed and models normal users executing approved programs.
- Workload
 - Small – An Android application package of size <100KB is requested for execution on the device. An individual service request is sent for an application while no other applications are executing. The number of requests sent is one and the frequency is once.
 - Medium – An Android application package of size >100KB and <500KB is requested for execution on the device. An individual service request is sent for an application while no other applications are executing. The number of requests sent is one and the frequency is once.

- Large – An Android application package of size $>500\text{KB}$ and $<1\text{MB}$ is requested for execution on the device. An individual service request is sent for an application while no other applications are executing. The number of requests sent is one and the frequency is once.
- Extra-Large – An Android application package of size $>1\text{MB}$ is requested for execution on the device. An individual service request is sent for an application while no other applications are executing. The number of request sent is one and the frequency is once.

- APS Switch
 - Enabled On – APS is fully functional on the device, checking each application package against the white-list prior to installation and execution.
 - Enabled Off – APS is turned off on the device, leaving application packages to be handled by the default Android security mechanisms.

3.9 Evaluation Technique

A combination of simulation and measurement is used to evaluate the system. The Android Software Development Kit (SDK) comes with a simulator for the Android environment. The simulator verifies that the protection mechanism compiles successfully on the Android device. Several Android OS versions are available for development use.

Following simulation, Android OS modifications are made and the kernel is compiled on an actual Android device. Thus, measurements are taken on the real-world system and directly show how the system performs.

The experimental configuration is an HTC Android Dev Phone 1 connected to a Dell Latitude D630 laptop. Modifications made to the Android OS and the modified kernel are compiled and flashed on the Dev Phone via the Dell laptop. Unapproved content is loaded on the phone via the Dell laptop. The combination of simulation and measurement supports each method being evaluated by the other.

3.10 Experimental Design

A full factorial design is used to evaluate the interaction between factors. The factors include application type, workload, and APS switch, with 3, 4, and 2 levels, respectively. This results in $3 \times 4 \times 2 = 24$ experiments. It is expected that sufficient statistical basis for analysis is achieved with 5 replications which brings the total number of experiments to 120. Each experiment is run until service requests are allowed or denied. The device, OS, onboard RAM, onboard storage, CPU, and default content all remain the same throughout the experiments while the factors vary.

The variance of the data in this research should be relatively low, as the results depend solely on either the allowing or blocking of application code. Results will be reported with 95% confidence. The system overhead is expected to be slightly higher with the APS employed and slightly higher with larger application packages.

3.11 Methodology Summary

Mobile devices are quickly becoming as popular as PCs for general purpose computing. While computer network-type protections are available for mobile networks, they are not nearly as sophisticated. This chapter describes the methodology for testing a mobile network protection mechanism for Google's Android platform installed on an

HTC Dev Phone. The goal of the research is to provide a protection mechanism within the kernel of the mobile device platform without incurring substantial system overhead.

The SUT and CUT are identified along with the accompanying parameters. Factors are selected from the system and workload parameters. The methodology varies these factors during experimentation to produce results in a variety of configurations. The metrics used for evaluation include percentage of unapproved content blocked, percentage of approved content allowed, and program delay.

The methodology consists of both simulation and measurement evaluations. The Android platform comes with an emulator used for the simulations and an HTC Dev Phone connected to a Dell Latitude D630 laptop is used for the measurement evaluations. A full factorial design is implemented with 120 experiments being conducted across 5 replications.

IV. Android Protection System Performance

4.1 Introduction

This chapter presents the Android Protection System (APS), a signed code modification of the Android OS 1.5 running on a smartphone device. White-list creation and hash digest placement are described in the security mechanism implementation. The evaluation technique is examined and results for functional protection and performance overhead are reported and analyzed.

4.2 Android Protection System Implementation

Proper identification of Android application code is essential for successful APS implementation. Android application code is delivered in packages called .jar or .apk files similar to .zip archives. Android applications are typically written in the Java programming language. The Dalvik Virtual Machine (DVM) operates strictly on Dalvik bytecode, so all Java bytecode is converted and stored in a file called Classes.dex, which is packaged inside the application-specific .apk file. The DVM must extract the Classes.dex file from the .apk to install and run the application.

Default applications come pre-installed on the Android device and all default applications are considered approved content. Other applications must go through the installation process prior to execution. If changes are made to application packages after installation, the application will not execute until it is reinstalled. During the installation process, APS computes a hash of the application package and compares the result to the white-list of approved content.

4.2.1 White-list Creation

The white-list stores a collection of content approved for execution on the device. Rather than storing exact copies of the application files, which would be highly inefficient, the APS computes a cryptographic hash digest offline for every approved application package. These digests are saved in a white-list for comparisons at runtime. These digests make it virtually impossible for an attacker to craft an application such that it would be allowed to execute on the system. Even if the digests are openly stored, the hashing is one-way, and it is impossible to compute a message from a digest.

The hashing algorithm used is the MD5 Message Digest Algorithm created and copyrighted by RSA Data Security. The algorithm can be viewed in its entirety in Appendix A. This algorithm comes complete with driver methods for creating hash digests from files, strings, or directly from standard input. The MDFile() method calculates hash digests for all approved .apk files offline. These digests make up the white-list used by the APS.

4.2.2 Hash Digest Placement

Once all hash digests are calculated, they are stored as Strings in a file within the Android kernel. Android applications are installed and loaded by a Package Manager. APS creates new functions within Android OS PackageManagerService which are accessible only by kernel-level processes, thus separating the protection mechanism from user space. The hash digests, cryptographic hash algorithm, and APS security function are placed within this service.

When a user attempts to install and execute an application on the Android device, the system jumps into the PackageManagerService routines. Before allowing an installation or runtime environment, the application package is supplied as input to a hashing function that returns the MD5 checksum of the file in the form of a hash digest. This digest is compared to the pre-computed values stored in the white-list. If a match is found, the application package is considered approved content, the packageApproved flag is set, and the application is allowed to install and execute on the device. If a match for the hash digest is not found in the white-list, the packageApproved flag is not set and the function sends an error message without installing the application or allowing it to execute on the device. Appendix B identifies modifications made to the Android 1.5 source code.

4.3 Evaluation Technique

To evaluate the protection performance of APS, it is tested with the protection mechanism both enabled and disabled. In the enabled configuration all application package installation requests first pass through the custom APS security function. Upon a white-list match the package is installed and a successful confirmation message is passed to the user. If there is no white-list match, the package is blocked from installation and a rejection message is passed to the user. The disabled configuration removes the custom APS security function; all application package installation requests are handled by the native Android system. In the disabled configuration it is expected all applications will be allowed to install and execute on the device. Section 4.4 examines results from protection testing.

To evaluate a system performance benchmark, the Android Debug Bridge (adb) measures program delay. The adb is a versatile tool for managing the state of an Android-powered device. It is a client-server program with built-in functions for timing metrics. The server component runs as a background process on the development machine (Dell laptop) and communicates with a daemon running as a background process on the Android device. The client on the development machine establishes communication through a command-line interface. A timestamp is taken when the system receives an installation request from the adb client. A second timestamp is taken when the installation request is finished processing and control is returned to the user. Elapsed time is reported in milliseconds. Section 4.4 examines results from performance testing.

4.4 Functional Protection Testing

Table 4.1 contains the functional protection results for the 120 tests conducted. The APS mechanism is enabled for the first 60 tests. 12 applications are individually submitted to the system for installation and 5 tests are run with each application for a total of 60 tests with this APS configuration. The same 60 tests are run with the APS mechanism disabled. The first letter of the application name identifies the size of the application package. Small, medium, large, and extra-large file sizes are represented by “s”, “m”, “l”, and “x” respectively. The remainder of the application name identifies the application type. Approved, non-malicious, and malicious types are represented by “app”, “non”, and “mal” respectively. Approved application packages have corresponding entries in the system white-list and are expected to be allowed. Non-

malicious application packages do not have corresponding entries in the white-list and are expected to be blocked. Malicious application packages are approved applications that have been modified by an attacker. These packages have corresponding entries in the white-list for the approved version, but the modified versions are expected to be blocked.

Table 4.1. APS Functional Protection Results

Tests	Application	Size (KB)	APS Status	Expected Action	Actual Action
1-5	s_app	82	On	Allow	Allow
6-10	m_app	360	On	Allow	Allow
11-15	l_app	999	On	Allow	Allow
16-20	x_app	1865	On	Allow	Allow
21-25	s_non	48	On	Block	Block
26-30	m_non	233	On	Block	Block
31-35	l_non	677	On	Block	Block
36-40	x_non	1465	On	Block	Block
41-45	s_mal	14	On	Block	Block
46-50	m_mal	209	On	Block	Block
51-55	l_mal	582	On	Block	Block
56-60	x_mal	1259	On	Block	Block
61-65	s_app	82	Off	Allow	Allow
66-70	m_app	360	Off	Allow	Allow
71-75	l_app	999	Off	Allow	Allow
76-80	x_app	1865	Off	Allow	Allow
81-85	s_non	48	Off	Allow	Allow
86-90	m_non	233	Off	Allow	Allow
91-95	l_non	677	Off	Allow	Allow
96-100	x_non	1465	Off	Allow	Allow
101-105	s_mal	14	Off	Allow	Allow
106-110	m_mal	209	Off	Allow	Allow
111-115	l_mal	582	Off	Allow	Allow
116-120	x_mal	1259	Off	Allow	Allow

Tests 1 through 20 evaluate the four approved application packages with APS enabled. In each case the actual action matches the expected action of “allow.” Tests 21 through 60 evaluate the four non-malicious and four malicious application packages, all of which should be blocked by APS. The results show that APS successfully produced the expected action in each case.

Tests 61 through 120 evaluate the 12 application packages against the system with APS disabled. The expected action for each of these tests is that the system will allow installation and execution. There is no mechanism outside user interaction in place to prevent installation of unapproved content. Test results indicate that the default Android protection mechanism produced the expected action for each test case.

APS is successful in preventing the execution of unapproved application packages on the Android device. 100% of unapproved content is blocked. APS is also successful in allowing approved application packages to execute on the Android device. 100% of approved content is allowed.

4.5 Performance Benchmark

The performance results of the 120 tests are shown in Table 4.2. Each row in the table represents a single test configuration that is repeated five times. The five test times are shown in the columns on the right with a calculated mean for each configuration. The table is organized by application name and size, making it simple to compare performance with APS enabled to performance with APS disabled (every row switches APS status). As expected, mean load times for test configurations with APS enabled are slightly higher than mean load times for test configurations with APS disabled.

Table 4.2. APS Performance Testing Results

Test Configuration			Timing Results (seconds)					
Application	Size (KB)	APS Status	Run 1	Run 2	Run 3	Run 4	Run 5	Mean
s_app	82	On	0.779	0.717	0.701	0.702	0.748	0.729
s_app	82	Off	0.733	0.686	0.733	0.701	0.686	0.708
m_app	360	On	2.886	3.057	2.995	2.871	2.698	2.901
m_app	360	Off	2.792	2.854	2.823	2.885	2.759	2.823
l_app	999	On	7.972	7.878	7.800	8.002	7.737	7.878
l_app	999	Off	7.768	7.862	7.519	7.670	7.846	7.733
x_app	1865	On	14.554	14.320	14.554	14.648	14.788	14.573
x_app	1865	Off	14.398	14.507	14.569	14.446	14.414	14.467
s_non	48	On	0.592	0.561	0.608	0.593	0.670	0.605
s_non	48	Off	0.608	0.561	0.639	0.592	0.608	0.602
m_non	233	On	2.120	1.900	1.965	1.777	1.856	1.924
m_non	233	Off	1.856	1.840	1.887	1.887	1.891	1.872
l_non	677	On	5.616	5.334	5.288	5.319	5.334	5.378
l_non	677	Off	5.155	5.194	5.350	5.147	5.396	5.248
x_non	1465	On	11.388	11.590	11.527	11.465	11.449	11.484
x_non	1465	Off	11.356	11.278	11.123	11.528	11.247	11.306
s_mal	14	On	0.203	0.186	0.171	0.187	0.171	0.184
s_mal	14	Off	0.124	0.156	0.192	0.173	0.171	0.163
m_mal	209	On	1.669	1.669	1.746	1.731	1.794	1.722
m_mal	209	Off	1.684	1.715	1.670	1.747	1.684	1.700
l_mal	582	On	4.586	4.679	4.680	4.539	4.452	4.587
l_mal	582	Off	4.662	4.555	4.554	4.554	4.477	4.560
x_mal	1259	On	9.843	9.930	10.046	9.937	9.828	9.917
x_mal	1259	Off	9.687	9.750	10.015	9.593	9.593	9.728

Figure 4.1 shows a linear response in application load time according to file size. The regression model for APS load time performance is

$$Load\ Time = 0.120 + 0.776 * File\ Size \quad (1)$$

where ‘File Size’ is the size of the application package file (.apk). The p value for the regression analysis is less than 0.001, providing convincing evidence that the regression model is a good fit for the data.

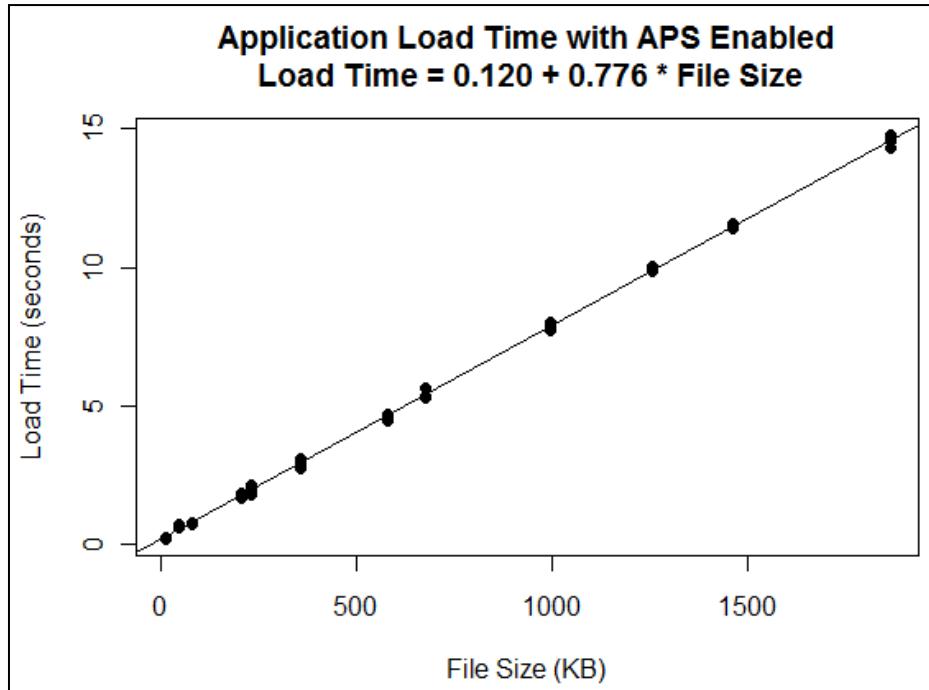


Figure 4.1. Application Load Time – APS Enabled

Figure 4.2 also shows a linear response in application load time according to file size. The regression model for default Android load time performance is

$$Load\ Time = 0.0947 + 0.776 * File\ Size \quad (2)$$

where ‘File Size’ is the size of the application package file (.apk). The only difference between these models is the location of the intercept.

The difference in performance mean times between APS enabled configurations and APS disabled configurations is minimal. Table 4.3 shows that the mean difference never exceeds 200 milliseconds, even for the largest application package sizes. The data shows a linear response in mean load time according to file size. The regression model for predicting the difference in mean time is

$$Difference = 24.978 + 0.086 * File\ Size \quad (3)$$

where ‘File Size’ is the size of the application package file (.apk). The 95% confidence interval for the intercept is [-15.107, 65.065] and for the slope is [0.040, 0.132].

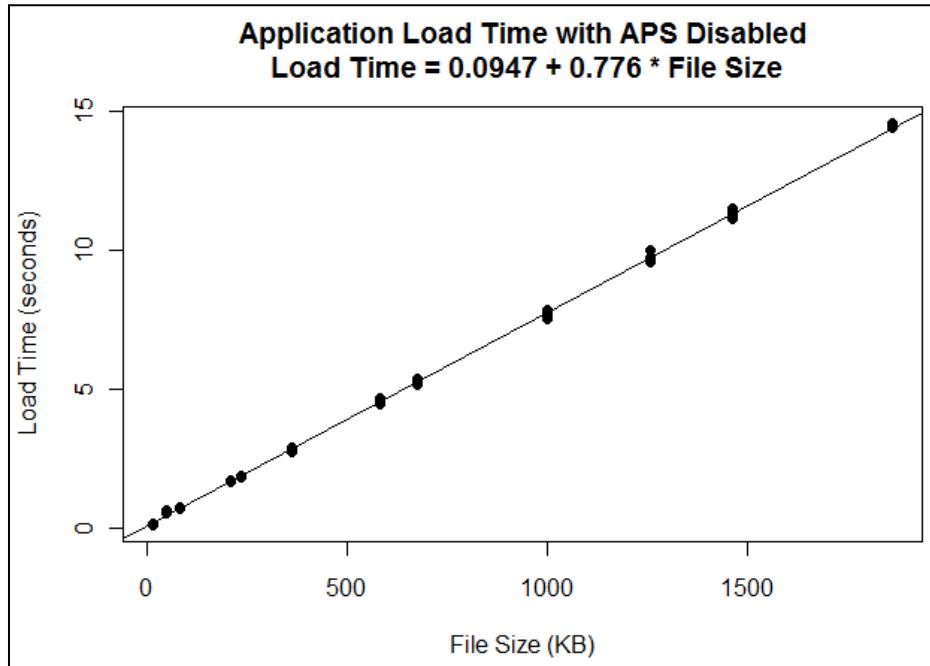


Figure 4.2. Application Load Time – APS Disabled

Table 4.3. Difference in Mean Load Times

Application	Size (KB)	APS Mean Time (sec)	Android Mean Time (sec)	Difference Mean Time (sec)
s_app	82	0.729	0.708	0.022
m_app	360	2.901	2.823	0.079
l_app	999	7.878	7.733	0.145
x_app	1865	14.573	14.467	0.106
s_non	48	0.605	0.602	0.003
m_non	233	1.924	1.872	0.051
l_non	677	5.378	5.248	0.130
x_non	1465	11.484	11.306	0.177
s_mal	14	0.184	0.163	0.020
m_mal	209	1.722	1.700	0.022
l_mal	582	4.587	4.560	0.027
x_mal	1259	9.917	9.728	0.189

The regression model is displayed in Figure 4.3. The plot has a few points outside the 95% confidence interval, but the model has a clear linear increase. The

largest application package sizes tested in this research approach 2MB, nearly double the 1MB average size for Android applications. Even at this large size, the mean difference in load time remains less than two-tenths of a second. The effect APS has on system performance remains unnoticeable to the user. For a user to notice a difference in system performance during an application installation process, the difference would have to be several seconds. This threshold is significantly larger than the 1 second threshold proposed by Nielsen [Nie93] because it takes place during the installation process, when users expect a delay. Using (3), an application package size would have to be 23MB in order for the user to experience an extra two seconds of delay with APS enabled. Thus, APS achieves the development goal of adding minimal performance overhead.

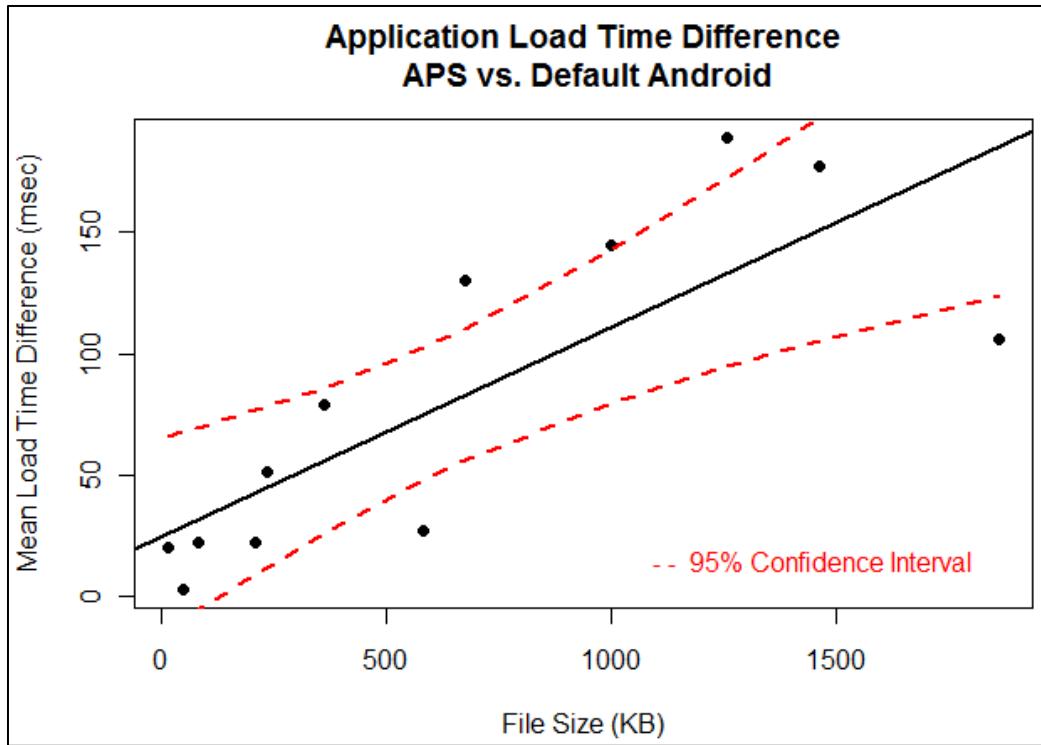


Figure 4.3. Difference in Application Load Times

4.6 Summary

APS performs very well with respect to the default Android protection mechanisms. Performance overhead for APS varies from 100.5% to 112.5% with respect to the default Android application installation process. The overhead is linearly increasing, but will remain within usable user limits for application packages up to at least 23MB. APS prevents 100% of unapproved installations while allowing 100% of approved content to install and execute. Chapter V addresses accomplishments of this research and proposes future work for adding capability to this protection mechanism.

V. Conclusions

5.1 Research Accomplishments

This research explores application protection in the smartphone world. Smartphones have greater capabilities than standard mobile phones, namely the ability to run third party applications. While this ability provides many conveniences to smartphone users, it also introduces new incentive and avenues for malicious activity. APS is a signed code security mechanism developed and implemented on an Android 1.5 kernel. APS focuses on protecting the mobile device from the installation and execution of unapproved applications, as this is where a high percentage of malicious activity originates. The performance of APS is compared to the performance of the default Android 1.5 platform.

Malicious applications can attack a mobile platform in many ways, but the application packages must be unpacked and installed on the device in order for internal code to execute. APS employs a security mechanism that hooks into the default application installation process on the Android platform. APS prevents applications from installing unless a hash digest computed during runtime matches a value stored in a white-list within the Android kernel.

APS blocked 100% of installation requests originating from unapproved content while allowing 100% of approved content to install and execute on the device. The APS security mechanism is implemented on the Android platform with little or no noticeable performance impact to the user. The mechanism is implemented during the installation process, so default and approved applications on the device continue to execute with no

performance impact. All performance impact is realized during application installation. A 1.8MB application package increases the system overhead during installation by 106 milliseconds. To reach two seconds in system overhead, a 23MB application package would be required (based on a 95% prediction interval). With an average application package size of ~1MB, the Android with APS does not impact user experience.

5.2 Research Impact

APS is perfectly suited for USAF and DoD enterprise deployment. However, the type of protection offered by APS would likely not appeal to a typical smartphone user. The signed-code mechanism of APS prevents users from downloading, installing, and executing unapproved applications. With thousands of applications available, typical users will avoid security mechanisms that hinder application freedom. Government, military, and many corporate organizations however, may welcome ways to control content allowed on company devices.

The white-list-based approach offered by APS is a simple means to an end for security-minded organizations. This research establishes a foundation for building a secure mobile device environment. Mobile system administrators can prepare a white-list of approved applications for company devices and use APS to ensure no additional applications are installed by device users. This security mechanism allows corporations to reap the benefits of smartphone technology without having to worry about creating vulnerabilities through the installation of unwanted applications.

5.3 Future Research Areas

The open source Android OS provides countless opportunities for custom security modifications. Thus, APS is not an end-all solution. Even so, this research presents a proof of concept that can be further developed into many types of security implementations.

APS focuses on application security. New application packages are verified against a white-list prior to installation. Protection is also needed for resident data on the mobile device. Signing static code does not prevent the signed code from executing in a malicious manner. Approved content that is later executed in some modified manner or order could gain access to sensitive data on the device that is assumed to be safe from attack. Adapting APS to ensure that applications execute as intended after installation would solve this problem.

The manifest file (AndroidManifest.xml) contained in each application package controls application permissions. All permissions are established at installation time and cannot be modified during runtime. The typical smartphone user has no insight into the legitimacy of application permission requests made during installation. Modifications could be made to the APS mechanism to examine the manifest file prior to installation, verifying that all requested permissions are necessary and legitimate. This approach removes user approval of any system permission access requests.

The previous section discussed the limitations APS imposes on smartphone application freedom. APS does however allow applications to be added to the device as long as the hash is verified against a white-list. This limits the application selection to only the pre-approved list. To approve more applications, the kernel has to be re-

compiled with a new white-list and flashed to the device. The APS security mechanism could be modified to allow new applications to be added on the fly. This solution would be difficult to develop while maintaining adequate protection, but success would make APS-enabled devices attractive to a much broader user base.

The current APS implementation hard-codes a white-list directly inside the custom security function. This is efficient, but security would be improved if the hash digests were stored in an encrypted file that was not opened until white-list values were requested. The performance hit taken for removing hard-coded white-list entries would be worthwhile for the improved security.

This research focuses on application security. There are many additional types of executable code resident on a mobile device. APS could be improved so as to protect against a wider range of file types. Applications have the ability to dynamically pull code from the Internet. This code would not pass through the application installation process, so APS would not block it. The challenge is developing a mechanism that protects against numerous file types and modes of execution without decreasing system performance and overloading limited resources on the mobile device.

Though developed on the Android platform, APS need not be a purely smartphone security mechanism. As Android is deployed to new types of devices, research opportunities continue to grow. Tablet PCs, desktops, laptops, and automotive computer systems are environments that need Android security research. APS has proven successful in a smartphone environment and should also be tested on more robust devices.

Appendix A. MD5 Message Digest Algorithm

Appendix A contains the MD5 Message Digest Algorithm as created and implemented by RSA Data Security, Inc. This code is used to calculate hash digests for approved Android application packages. These hash digests are stored in a white-list for use in the APS implementation.

```
/*
*****
** md5.h -- Header file for implementation of MD5
**
** RSA Data Security, Inc. MD5 Message Digest Algorithm
**
** Created: 2/17/90 RLR
**
** Revised: 12/27/90 SRD,AJ,BSK,JT Reference C version
**
** Revised (for MD5): RLR 4/27/91
**
** -- G modified to have y&~z instead of y&z
**
** -- FF, GG, HH modified to add in last register done
**
** -- Access pattern: round 2 works mod 5, round 3 works mod 3
**
** -- distinct additive constant for each step
**
** -- round 4 added, working mod 7
**

*****
*/
/* *****
```

```
*****
** Copyright (C) 1990, RSA Data Security, Inc. All rights
reserved. **
**
```

```

** License to copy and use this software is granted provided
that **
** it is identified as the "RSA Data Security, Inc. MD5 Message
**
** Digest Algorithm" in all material mentioning or referencing
this **
** software or this function.
**
**
** License is also granted to make and use derivative works
**
** provided that such works are identified as "derived from the
RSA **
** Data Security, Inc. MD5 Message Digest Algorithm" in all
**
** material mentioning or referencing the derived work.
**
**
** RSA Data Security, Inc. makes no representations concerning
**
** either the merchantability of this software or the
suitability **
** of this software for any particular purpose. It is provided
"as **
** is" without express or implied warranty of any kind.
**
**
** These notices must be retained in any copies of any part of
this **
** documentation and/or software.
**

*****
*/
/* typedef a 32 bit type */
typedef unsigned long int UINT4;

/* Data structure for MD5 (Message Digest) computation */
typedef struct {
    UINT4 i[2];                                /* number of _bits_ handled mod
2^64 */
    UINT4 buf[4];                                /* scratch
buffer */
    unsigned char in[64];                         /* input
buffer */

```

```

    unsigned char digest[16];      /* actual digest after MD5Final
call */
} MD5_CTX;

void MD5Init ();
void MD5Update ();
void MD5Final ();

/*
*****
** End of md5.h
**
***** (cut)
*****
*/
/*
*****
** md5.c
**
** RSA Data Security, Inc. MD5 Message Digest Algorithm
**
** Created: 2/17/90 RLR
**
** Revised: 1/91 SRD,AJ,BSK,JT Reference C Version
**
*****
*/
/*
*****
** Copyright (C) 1990, RSA Data Security, Inc. All rights
reserved. */
**
** License to copy and use this software is granted provided
that */
** it is identified as the "RSA Data Security, Inc. MD5 Message
**
** Digest Algorithm" in all material mentioning or referencing
this */

```

```

** software or this function.
**
**
**
** License is also granted to make and use derivative works
**
** provided that such works are identified as "derived from the
RSA **
** Data Security, Inc. MD5 Message Digest Algorithm" in all
**
** material mentioning or referencing the derived work.
**
**
**
** RSA Data Security, Inc. makes no representations concerning
**
** either the merchantability of this software or the
suitability **
** of this software for any particular purpose. It is provided
"as **
** is" without express or implied warranty of any kind.
**
**
**
** These notices must be retained in any copies of any part of
this **
** documentation and/or software.
**

*****
*/

```

```

/* -- include the following line if the md5.h header file is
separate -- */
/* #include "md5.h" */

/* forward declaration */
static void Transform ();

static unsigned char PADDING[64] = {
    0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

```

```

/* F, G and H are basic MD5 functions: selection, majority,
parity */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n)))))

/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4
*/
/* Rotation is separate from addition to prevent recomputation */
#define FF(a, b, c, d, x, s, ac) \
{ (a) += F ((b), (c), (d)) + (x) + (UINT4) (ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
#define GG(a, b, c, d, x, s, ac) \
{ (a) += G ((b), (c), (d)) + (x) + (UINT4) (ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
#define HH(a, b, c, d, x, s, ac) \
{ (a) += H ((b), (c), (d)) + (x) + (UINT4) (ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}
#define II(a, b, c, d, x, s, ac) \
{ (a) += I ((b), (c), (d)) + (x) + (UINT4) (ac); \
  (a) = ROTATE_LEFT ((a), (s)); \
  (a) += (b); \
}

void MD5Init (mdContext)
MD5_CTX *mdContext;
{
    mdContext->i[0] = mdContext->i[1] = (UINT4) 0;

    /* Load magic initialization constants.
     */
    mdContext->buf[0] = (UINT4) 0x67452301;
    mdContext->buf[1] = (UINT4) 0xefcdab89;
    mdContext->buf[2] = (UINT4) 0x98badcfe;
    mdContext->buf[3] = (UINT4) 0x10325476;
}

void MD5Update (mdContext, inBuf, inLen)
MD5_CTX *mdContext;

```

```

unsigned char *inBuf;
unsigned int inLen;
{
    UINT4 in[16];
    int mdi;
    unsigned int i, ii;

    /* compute number of bytes mod 64 */
    mdi = (int)((mdContext->i[0] >> 3) & 0x3F);

    /* update number of bits */
    if ((mdContext->i[0] + ((UINT4)inLen << 3)) < mdContext->i[0])
        mdContext->i[1]++;
    mdContext->i[0] += ((UINT4)inLen << 3);
    mdContext->i[1] += ((UINT4)inLen >> 29);

    while (inLen--) {
        /* add new character to buffer, increment mdi */
        mdContext->in[mdi++] = *inBuf++;

        /* transform if necessary */
        if (mdi == 0x40) {
            for (i = 0, ii = 0; i < 16; i++, ii += 4)
                in[i] = (((UINT4)mdContext->in[ii+3]) << 24) |
                    (((UINT4)mdContext->in[ii+2]) << 16) |
                    (((UINT4)mdContext->in[ii+1]) << 8) |
                    ((UINT4)mdContext->in[ii]);
            Transform (mdContext->buf, in);
            mdi = 0;
        }
    }
}

void MD5Final (mdContext)
MD5_CTX *mdContext;
{
    UINT4 in[16];
    int mdi;
    unsigned int i, ii;
    unsigned int padLen;

    /* save number of bits */
    in[14] = mdContext->i[0];
    in[15] = mdContext->i[1];

    /* compute number of bytes mod 64 */
    mdi = (int)((mdContext->i[0] >> 3) & 0x3F);

    /* pad out to 56 mod 64 */
    padLen = (mdi < 56) ? (56 - mdi) : (120 - mdi);
}

```

```

MD5Update (mdContext, PADDING, padLen);

/* append length in bits and transform */
for (i = 0, ii = 0; i < 14; i++, ii += 4)
    in[i] = (((UINT4)mdContext->in[ii+3]) << 24) |
        (((UINT4)mdContext->in[ii+2]) << 16) |
        (((UINT4)mdContext->in[ii+1]) << 8) |
        ((UINT4)mdContext->in[ii]);
Transform (mdContext->buf, in);

/* store buffer in digest */
for (i = 0, ii = 0; i < 4; i++, ii += 4) {
    mdContext->digest[ii] = (unsigned char) (mdContext->buf[i] &
0xFF);
    mdContext->digest[ii+1] =
        (unsigned char) ((mdContext->buf[i] >> 8) & 0xFF);
    mdContext->digest[ii+2] =
        (unsigned char) ((mdContext->buf[i] >> 16) & 0xFF);
    mdContext->digest[ii+3] =
        (unsigned char) ((mdContext->buf[i] >> 24) & 0xFF);
}
}

/* Basic MD5 step. Transform buf based on in.
 */
static void Transform (buf, in)
UINT4 *buf;
UINT4 *in;
{
    UINT4 a = buf[0], b = buf[1], c = buf[2], d = buf[3];

    /* Round 1 */
#define S11 7
#define S12 12
#define S13 17
#define S14 22
    FF ( a, b, c, d, in[ 0], S11, 3614090360); /* 1 */
    FF ( d, a, b, c, in[ 1], S12, 3905402710); /* 2 */
    FF ( c, d, a, b, in[ 2], S13, 606105819); /* 3 */
    FF ( b, c, d, a, in[ 3], S14, 3250441966); /* 4 */
    FF ( a, b, c, d, in[ 4], S11, 4118548399); /* 5 */
    FF ( d, a, b, c, in[ 5], S12, 1200080426); /* 6 */
    FF ( c, d, a, b, in[ 6], S13, 2821735955); /* 7 */
    FF ( b, c, d, a, in[ 7], S14, 4249261313); /* 8 */
    FF ( a, b, c, d, in[ 8], S11, 1770035416); /* 9 */
    FF ( d, a, b, c, in[ 9], S12, 2336552879); /* 10 */
    FF ( c, d, a, b, in[10], S13, 4294925233); /* 11 */
    FF ( b, c, d, a, in[11], S14, 2304563134); /* 12 */
    FF ( a, b, c, d, in[12], S11, 1804603682); /* 13 */
    FF ( d, a, b, c, in[13], S12, 4254626195); /* 14 */
}

```

```

FF ( c, d, a, b, in[14], S13, 2792965006); /* 15 */
FF ( b, c, d, a, in[15], S14, 1236535329); /* 16 */

/* Round 2 */
#define S21 5
#define S22 9
#define S23 14
#define S24 20
    GG ( a, b, c, d, in[ 1], S21, 4129170786); /* 17 */
    GG ( d, a, b, c, in[ 6], S22, 3225465664); /* 18 */
    GG ( c, d, a, b, in[11], S23, 643717713); /* 19 */
    GG ( b, c, d, a, in[ 0], S24, 3921069994); /* 20 */
    GG ( a, b, c, d, in[ 5], S21, 3593408605); /* 21 */
    GG ( d, a, b, c, in[10], S22, 38016083); /* 22 */
    GG ( c, d, a, b, in[15], S23, 3634488961); /* 23 */
    GG ( b, c, d, a, in[ 4], S24, 3889429448); /* 24 */
    GG ( a, b, c, d, in[ 9], S21, 568446438); /* 25 */
    GG ( d, a, b, c, in[14], S22, 3275163606); /* 26 */
    GG ( c, d, a, b, in[ 3], S23, 4107603335); /* 27 */
    GG ( b, c, d, a, in[ 8], S24, 1163531501); /* 28 */
    GG ( a, b, c, d, in[13], S21, 2850285829); /* 29 */
    GG ( d, a, b, c, in[ 2], S22, 4243563512); /* 30 */
    GG ( c, d, a, b, in[ 7], S23, 1735328473); /* 31 */
    GG ( b, c, d, a, in[12], S24, 2368359562); /* 32 */

/* Round 3 */
#define S31 4
#define S32 11
#define S33 16
#define S34 23
    HH ( a, b, c, d, in[ 5], S31, 4294588738); /* 33 */
    HH ( d, a, b, c, in[ 8], S32, 2272392833); /* 34 */
    HH ( c, d, a, b, in[11], S33, 1839030562); /* 35 */
    HH ( b, c, d, a, in[14], S34, 4259657740); /* 36 */
    HH ( a, b, c, d, in[ 1], S31, 2763975236); /* 37 */
    HH ( d, a, b, c, in[ 4], S32, 1272893353); /* 38 */
    HH ( c, d, a, b, in[ 7], S33, 4139469664); /* 39 */
    HH ( b, c, d, a, in[10], S34, 3200236656); /* 40 */
    HH ( a, b, c, d, in[13], S31, 681279174); /* 41 */
    HH ( d, a, b, c, in[ 0], S32, 3936430074); /* 42 */
    HH ( c, d, a, b, in[ 3], S33, 3572445317); /* 43 */
    HH ( b, c, d, a, in[ 6], S34, 76029189); /* 44 */
    HH ( a, b, c, d, in[ 9], S31, 3654602809); /* 45 */
    HH ( d, a, b, c, in[12], S32, 3873151461); /* 46 */
    HH ( c, d, a, b, in[15], S33, 530742520); /* 47 */
    HH ( b, c, d, a, in[ 2], S34, 3299628645); /* 48 */

/* Round 4 */
#define S41 6
#define S42 10

```

```

#define S43 15
#define S44 21
    II ( a, b, c, d, in[ 0], S41, 4096336452); /* 49 */
    II ( d, a, b, c, in[ 7], S42, 1126891415); /* 50 */
    II ( c, d, a, b, in[14], S43, 2878612391); /* 51 */
    II ( b, c, d, a, in[ 5], S44, 4237533241); /* 52 */
    II ( a, b, c, d, in[12], S41, 1700485571); /* 53 */
    II ( d, a, b, c, in[ 3], S42, 2399980690); /* 54 */
    II ( c, d, a, b, in[10], S43, 4293915773); /* 55 */
    II ( b, c, d, a, in[ 1], S44, 2240044497); /* 56 */
    II ( a, b, c, d, in[ 8], S41, 1873313359); /* 57 */
    II ( d, a, b, c, in[15], S42, 4264355552); /* 58 */
    II ( c, d, a, b, in[ 6], S43, 2734768916); /* 59 */
    II ( b, c, d, a, in[13], S44, 1309151649); /* 60 */
    II ( a, b, c, d, in[ 4], S41, 4149444226); /* 61 */
    II ( d, a, b, c, in[11], S42, 3174756917); /* 62 */
    II ( c, d, a, b, in[ 2], S43, 718787259); /* 63 */
    II ( b, c, d, a, in[ 9], S44, 3951481745); /* 64 */

    buf[0] += a;
    buf[1] += b;
    buf[2] += c;
    buf[3] += d;
}

/*
*****
** End of md5.c
**
***** (cut)
*****
*/
/*
*****
** md5driver.c -- sample routines to test
**
** RSA Data Security, Inc. MD5 message digest algorithm.
**
** Created: 2/16/90 RLR
**
** Updated: 1/91 SRD
**
***** */

```

```

*/
/*
*****
** Copyright (C) 1990, RSA Data Security, Inc. All rights
reserved. **
**
** RSA Data Security, Inc. makes no representations concerning
**
** either the merchantability of this software or the
suitability   **
** of this software for any particular purpose. It is provided
"as   **
** is" without express or implied warranty of any kind.
**
**
** These notices must be retained in any copies of any part of
this   **
** documentation and/or software.
**

*****
*/
#include <stdio.h>
#include <sys/types.h>
#include <time.h>
#include <string.h>
/* -- include the following file if the file md5.h is separate --
*/
/* #include "md5.h" */

/* Prints message digest buffer in mdContext as 32 hexadecimal
digits.
   Order is from low-order byte to high-order byte of digest.
   Each byte is printed with high-order hexadecimal digit first.
*/
static void MDPrint (mdContext)
MD5_CTX *mdContext;
{
    int i;

    for (i = 0; i < 16; i++)
        printf ("%02x", mdContext->digest[i]);
}

```

```

/* size of test block */
#define TEST_BLOCK_SIZE 1000

/* number of blocks to process */
#define TEST_BLOCKS 10000

/* number of test bytes = TEST_BLOCK_SIZE * TEST_BLOCKS */
static long TEST_BYTES = (long)TEST_BLOCK_SIZE *
(long)TEST_BLOCKS;

/* A time trial routine, to measure the speed of MD5.
   Measures wall time required to digest TEST_BLOCKS *
TEST_BLOCK_SIZE
   characters.

*/
static void MDTimeTrial ()
{
    MD5_CTX mdContext;
    time_t endTime, startTime;
    unsigned char data[TEST_BLOCK_SIZE];
    unsigned int i;

    /* initialize test data */
    for (i = 0; i < TEST_BLOCK_SIZE; i++)
        data[i] = (unsigned char)(i & 0xFF);

    /* start timer */
    printf ("MD5 time trial. Processing %ld characters...\n",
TEST_BYTES);
    time (&startTime);

    /* digest data in TEST_BLOCK_SIZE byte blocks */
    MD5Init (&mdContext);
    for (i = TEST_BLOCKS; i > 0; i--)
        MD5Update (&mdContext, data, TEST_BLOCK_SIZE);
    MD5Final (&mdContext);

    /* stop timer, get time difference */
    time (&endTime);
    MDPrint (&mdContext);
    printf (" is digest of test input.\n");
    printf
        ("Seconds to process test input: %ld\n", (long)(endTime-
startTime));
    printf
        ("Characters processed per second: %ld\n",
TEST_BYTES/(endTime-startTime));
}

```

```

/* Computes the message digest for string inString.
   Prints out message digest, a space, the string (in quotes) and
a
   carriage return.
*/
static void MDString (inString)
char *inString;
{
    MD5_CTX mdContext;
    unsigned int len = strlen (inString);

    MD5Init (&mdContext);
    MD5Update (&mdContext, inString, len);
    MD5Final (&mdContext);
    MDPrint (&mdContext);
    printf (" \"%s\"\n\n", inString);
}

/* Computes the message digest for a specified file.
   Prints out message digest, a space, the file name, and a
carriage
   return.
*/
static void MDFile (filename)
char *filename;
{
    FILE *inFile = fopen (filename, "rb");
    MD5_CTX mdContext;
    int bytes;
    unsigned char data[1024];

    if (inFile == NULL) {
        printf ("%s can't be opened.\n", filename);
        return;
    }

    MD5Init (&mdContext);
    while ((bytes = fread (data, 1, 1024, inFile)) != 0)
        MD5Update (&mdContext, data, bytes);
    MD5Final (&mdContext);
    MDPrint (&mdContext);
    printf ("%s\n", filename);
    fclose (inFile);
}

/* Writes the message digest of the data from stdin onto stdout,
   followed by a carriage return.
*/
static void MDFilter ()
{

```

```

MD5_CTX mdContext;
int bytes;
unsigned char data[16];

MD5Init (&mdContext);
while ((bytes = fread (data, 1, 16, stdin)) != 0)
    MD5Update (&mdContext, data, bytes);
MD5Final (&mdContext);
MDPrint (&mdContext);
printf ("\n");
}

/* Runs a standard suite of test data.
 */
static void MDTestSuite ()
{
    printf ("MD5 test suite results:\n\n");
    MDString ("");
    MDString ("a");
    MDString ("abc");
    MDString ("message digest");
    MDString ("abcdefghijklmnopqrstuvwxyz");
    MDString

("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
}
    MDString
    ("1234567890123456789012345678901234567890\
1234567890123456789012345678901234567890");
/* Contents of file foo are "abc" */
MDFile ("foo");
}

void main (argc, argv)
int argc;
char *argv[];
{
    int i;

    /* For each command line argument in turn:
     ** filename          -- prints message digest and name of file
     ** -sstring          -- prints message digest and contents of
     string
     ** -t                -- prints time trial statistics for 1M
     characters
     ** -x                -- execute a standard suite of test data
     ** (no args)         -- writes messages digest of stdin onto
     stdout
     */
    if (argc == 1)

```

```

        MDFilter ();
else
    for (i = 1; i < argc; i++)
        if (argv[i][0] == '-' && argv[i][1] == 's')
            MDString (argv[i] + 2);
        else if (strcmp (argv[i], "-t") == 0)
            MDTIMETrial ();
        else if (strcmp (argv[i], "-x") == 0)
            MDTestSuite ();
        else MDFile (argv[i]);
}

/*
*****
*** End of md5driver.c
**
***** (cut)
*****
*/

```

Appendix B. APS Modification to Android OS 1.5

Appendix B contains the APS security mechanism implementation on the Android OS 1.5. APS modifies the contents of PackageManagerService.java. All APS modifications fall within the installPackageLI() function shown here. The PackageManagerService.java file in its entirety can be viewed in the com.android.server package downloaded from <http://source.android.com>.

```
/*
 * Copyright (C) 2006 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the
 * "License");
 * you may not use this file except in compliance with the
 * License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software
 * distributed under the License is distributed on an "AS IS"
 * BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing
 * permissions and
 * limitations under the License.
 */

package com.android.server;

private PackageInstalledInfo installPackageLI(Uri pPackageURI,
                                              int pFlags, boolean newInstall) {
    File tmpPackageFile = null;
    String pkgName = null;
    boolean forwardLocked = false;
    boolean replacingExistingPackage = false;
    // Result object to be returned
    PackageInstalledInfo res = new PackageInstalledInfo();
    res.returnValue = PackageManager.INSTALL_SUCCEEDED;
    res.uid = -1;
```

```

res.pkg = null;
res.removedInfo = new PackageRemovedInfo();

main_flow: try {
    tmpPackageFile = createTempPackageFile();
    if (tmpPackageFile == null) {
        res.returnValue =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
        break main_flow;
    }
    tmpPackageFile.deleteOnExit(); // paranoia
    if (pPackageURI.getScheme().equals("file")) {
        final File srcPackageFile = new
File(pPackageURI.getPath());
            // We copy the source package file to a temp file
and then rename it to the
            // destination file in order to eliminate a
window where the package directory
            // scanner notices the new package file but it's
not completely copied yet.
        if (!FileUtils.copyWithFile(srcPackageFile,
tmpPackageFile)) {
            Log.e(TAG, "Couldn't copy package file to
temp file.");
            res.returnValue =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
            break main_flow;
        }
    } else if (pPackageURI.getScheme().equals("content"))
{
    ParcelFileDescriptor fd;
    try {
        fd =
mContext.getContentResolver().openFileDescriptor(pPackageURI,
"x");
    } catch (FileNotFoundException e) {
        Log.e(TAG, "Couldn't open file descriptor
from download service.");
        res.returnValue =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
        break main_flow;
    }
    if (fd == null) {
        Log.e(TAG, "Couldn't open file descriptor
from download service (null).");
        res.returnValue =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
        break main_flow;
    }
    if (Config.LOGV) {

```

```

        Log.v(TAG, "Opened file descriptor from
download service.");
    }
    ParcelFileDescriptor.AutoCloseInputStream
    dlStream = new
ParcelFileDescriptor.AutoCloseInputStream(fd);
    // We copy the source package file to a temp file
and then rename it to the
    // destination file in order to eliminate a
window where the package directory
    // scanner notices the new package file but it's
not completely copied yet.
    if (!FileUtils.copyToFile(dlStream,
tmpPackageFile)) {
        Log.e(TAG, "Couldn't copy package stream to
temp file.");
        res.returnCode =
PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE;
        break main_flow;
    }
} else {
    Log.e(TAG, "Package URI is not 'file:' or
'content:' - " + pPackageURI);
    res.returnCode =
PackageManager.INSTALL_FAILED_INVALID_URI;
    break main_flow;
}
pkgName = PackageParser.parsePackageName(
    tmpPackageFile.getAbsolutePath(), 0);
if (pkgName == null) {
    Log.e(TAG, "Couldn't find a package name in : " +
tmpPackageFile);
    res.returnCode =
PackageManager.INSTALL_FAILED_INVALID_APK;
    break main_flow;
}
res.name = pkgName;
//initialize some variables before installing pkg
final String pkgFileName = pkgName + ".apk";

/*
 * Android Protection System (APS) Code added by Capt
Jonathan D. Stueckle, USAF
 * Air Force Institute of Technology Graduate Student,
March 2011
 *
 * A boolean flag 'packageApproved' is initialized for
storing the result of the content hashing

```

```

        * and comparison. The flag remains false unless the
value is updated based on the result from
        * the custom approveApk() function.
        *
        * The full path of the application package requesting
installation is sent to the custom function.
        * This source file is run through the hashing algorithm
and returns a boolean "true" if a match
        * is found. In this case, the application package then
continues through the installation process.
        * If no match is found, the function exits leaving the
flag with a "false" value, causing the
        * installPackageLI() function to exit with an "Invalid
APK" Error message.
        */

        /* Flag to store hashing & comparison result (added
JDS) */
        boolean packageApproved = false;

        /* Full application package path sent to custom APS
function (added JDS) */
        packageApproved=approveApk(tmpPackageFile.getPath());

        /*
        * Hash digest match was found in white-list, so
application package is approved
        * and can continue through normal installation
process. (added JDS)
        */
        if(packageApproved)
        {

            /* Normal installation process not modified by
JDS */
            final File destDir =
((pFlags&PackageManager.FORWARD_LOCK_PACKAGE) != 0)
                ? mDrmAppPrivateInstallDir
                    : mAppInstallDir;
            final File destPackageFile = new File(destDir,
pkgFileName);
            final String destFilePath =
destPackageFile.getAbsolutePath();
            File destResourceFile;
            if ((pFlags&PackageManager.FORWARD_LOCK_PACKAGE)
!= 0) {
                final String publicZipFileName = pkgName +
".zip";
                destResourceFile = new File(mAppInstallDir,
publicZipFileName);
            }
        }
    }
}

```

```

        forwardLocked = true;
    } else {
        destResourceFile = destPackageFile;
    }
    // Retrieve PackageSettings and parse package
    int parseFlags = PackageParser.PARSE_CHATTY;
    parseFlags |= mDefParseFlags;
    PackageParser pp = new
    PackageParser(tmpPackageFile.getPath());
        pp.setSeparateProcesses(mSeparateProcesses);
        pp.setSdkVersion(mSdkVersion);
        final PackageParser.Package pkg =
    pp.parsePackage(tmpPackageFile,
                    destPackageFile.getAbsolutePath(),
mMetrics, parseFlags);
        if (pkg == null) {
            res.returnCode = pp.getParseError();
            break main_flow;
        }
        if (GET_CERTIFICATES &&
!pp.collectCertificates(pkg, parseFlags)) {
            res.returnCode = pp.getParseError();
            break main_flow;
        }

        synchronized (mPackages) {
            //check if installing already existing
package
            if
((pFlags&PackageManager.REPLACE_EXISTING_PACKAGE) != 0
&&
mPackages.containsKey(pkgName)) {
                replacingExistingPackage = true;
            }
        }

        if(replacingExistingPackage) {
            replacePackageLI(pkgName,
                            tmpPackageFile,
                            destFilePath, destPackageFile,
destResourceFile,
                            pkg, forwardLocked, newInstall,
                            res);
        } else {
            installNewPackageLI(pkgName,
                            tmpPackageFile,
                            destFilePath, destPackageFile,
destResourceFile,
                            pkg, forwardLocked, newInstall,
                            res);
        }
    }
}

```

```

        }
    } //End of normal installation process

    /*
     * Hash digest match was not found in white-list, so
     application package is not
     * approved and is blocked from continuing through the
     installation process.
     * APS adds an error message here to indicate to the
     user that the application
     * package was not approved. (added JDS)
     */
    else {
        /* Message sent back to user (added JDS) */
        res.returnValue =
PackageManager.INSTALL_FAILED_INVALID_APK;
        /*Skip installation process and return error
(added JDS) */
        break main_flow;
    }

} finally {
    if (tmpPackageFile != null && tmpPackageFile.exists())
{
    tmpPackageFile.delete();
}
}
return res;
}

/*
 * Android Protection System (APS) Code added by Capt
Jonathan D. Stueckle, USAF
 * Air Force Institute of Technology Graduate Student, March
2011
 *
 * This code was obtained from http://www.apache.org and
provides the
 * functionality for calculating MD5 hash digests for the
application packages.
 * The APS custom approveApk() function utilizes
getMD5Checksum(), which then calls
 * createChecksum() to help calculate the hash.
 *
 * getMD5Checksum() incorporates a fast way to convert a byte
array to a HEX string.
 */
public static String getMD5Checksum(String filename) throws
Exception {

```

```

    /* byte array to hold result from hashing function (added
JDS) */
    byte[] b = createChecksum(filename);

    /* String to hold HEX conversion of byte array (added JDS)
*/
    String result = "";

    /* Convert byte array to HEX string (added JDS) */
    for (int i=0; i < b.length; i++) {
        result +=
            Integer.toHexString( ( b[i] & 0xff ) + 0x100,
16).substring( 1 );
    }
    /* Hash digest of application package now returned to
approveApk() function
     * for determination of white-list match. (added JDS)
     */
    return result;
}

public static byte[] createChecksum(String filename) throws
Exception {

    /* Read in application package for hashing (added JDS) */
    InputStream fis = new FileInputStream(filename);

    /* Calculate MD5 checksum of application package (added
JDS) */
    byte[] buffer = new byte[1024];
    MessageDigest complete = MessageDigest.getInstance("MD5");
    int numRead;
    do {
        numRead = fis.read(buffer);
        if (numRead > 0) {
            complete.update(buffer, 0, numRead);
        }
    } while (numRead != -1);
    fis.close();
    /* Returns byte array containing hash digest (added JDS)
*/
    return complete.digest();
}

/*
*

```

```

        * Android Protection System (APS) Code written by Capt
Jonathan D. Stueckle, USAF
        * Air Force Institute of Technology Graduate Student, March
2011
        *
        * This function receives the full path of an application
packages as input and
        * returns a boolean output indicating if a hash digest match
was found in the
        * white-list. A value of 'true' corresponds to a match, so
the boolean
        * 'approvedContent' is initialized to 'false.'
        *
        * The APS white-list consists of hash digests stored in
strings. There is a digest
        * corresponding to each default application on the system as
well as for external
        * applications that have been approved.
        *
        * A string 'hash_check' is set to the return value from the
getMD5Checksum() function.
        * This string is a HEX representation of the MD5 hash digest
computed on the
        * submitted application package.
        *
        * This string is then compared against each string stored in
the white-list. If a match
        * is found, the boolean flag is set to 'true' and returned.
This allows the submitted
        * application package to continue through the installation
process and the application
        * be allowed to execute on the Android device. If no match
is found, the flag remains
        * 'false' and when returned it blocks the application
package from installation, denying
        * execution of the application on the device.
        */

```

```

public static boolean approveApk(String filePath) {

    /* Flag to store hashing & comparison result */
    boolean approvedContent = false;

    /* APS white-list - represents all approved applications */
    String alarm = "8896f8d227b04781daaf095c3167736d";
    //Default app
    String browser = "a3f878fc3450f69543bd689f148a6cd7";
    //Default app
    String calc = "af7704733992987922f89e4d09607def";
    //Default app

```

```

String calendar = "c7a5ba3b1b03fcdad1bfbdAAA2588a6a";
//Default app
String calProv = "e3c570a2f83e2dd73610b0c42c3ec1cb";
//Default app
String camera = "2108c148abb3c9ecfacc3d420359f267";
//Default app
String contacts = "7207e0d85e4655da4486b9febb0d2da5";
//Default app
String contProv = "7d76c5495830aed6c9855a37e5d58cd5";
//Default app
String dev = "b682a76a5de57d2399136d0739e3f4f6";
//Default app
String downProv = "ab73085f4bca1720021b9766d4930d07";
//Default app
String drmProv = "2a320eef517fe3213333f639e8ce498";
//Default app
String email = "f1a7e8a24fc6492a873af6939bb103a2";
//Default app
String googSearch = "06a45a35afb6efebecd295255357ba93";
//Default app
String html = "46db4ef1e3b005c8313d11e83e0955af";
//Default app
String imProv = "7629ec3578803de655508718c9e5f577";
//Default app
String latin = "330335bfc4c3333fa8d15e77caebea90";
//Default app
String launch = "23491fcbd1d40701e9bbfc1be53a7af9";
//Default app
String media = "7339bd4c3e81fb7a43601ce05c280ba3";
//Default app
String mms = "e81e0161aabdcfe7df30824f4763b9f";
//Default app
String music = "60dc8e68ec2a850e34c61e459e4d4304";
//Default app
String packIns = "1c5355e767f1887a725d555b38a5abd0";
//Default app
String phone = "a42e096375cb31eb2cdc733818300607";
//Default app
String sdk = "6ba40c5241c04d395b5303c8f7dd2aab";
//Default app
String settings = "e25d7d5f97cac8650c10a9e5ab3faa30";
//Default app
String setProv = "6abe6e95a75a78ec122714a889d9878a";
//Default app
String sound = "b3226a2a4ef823b71c738a87e366a0db";
//Default app
String spare = "b7ebad29f535b4645f58f013cbe86bd7";
//Default app
String subsc = "282f7575bcc9bbfb6570e060635d58a3";
//Default app

```

```

String telep = "893d234aac0fcde18b0246856d2cb455";
//Default app
String term = "16e003868e775a18805955e7ffe736e4";
//Default app
String user = "69cf79ae88262974b7e194ba91d3476d";
//Default app
String voice = "4c8a08b5c4c1bbb653d8d8eba72a6be3";
//Default app
String framres = "1bfdef7219657c99badbc593851ed4a7";
//Default app
String appinst = "5ebbe69c85dbe29f26faa51dbc02f730";
//AppsInstaller
String algtut = "dc8c5fe4bd847b42a47664a989a9932c";
//AlgebraTutor
String mictagread = "3e0b5702d3fb27af68d1d701b9cc9a14";
//MicrosoftTagReader
String tinyflash = "5be80ad02692a41912aaeb2d01c2a8c9";
//TinyFlashlight
String apricalc = "0deded4890d8dde529761d16ea75d3f01";
//aPriceCalc
String autokill = "bcd8b0a97b7750da7c2c19a1d39b09fe";
//AutorunKiller
String ligrac = "4bdee01df1fc194cb014a96653b75096";
//LightRacer
String vidbox = "c1f012df7d3d463b2c67cc6bc70d82cf";
//videobox

/* Variable to store hash digest of application package */
String hash_check = "";

try{
    /* Get hash digest */
    hash_check = getMD5Checksum(filePath);
}
catch (Exception e) {
    e.printStackTrace();
}
/* Compare hash digest to all values stored in the APS
white-list */

if(hash_check.equals(voice) || hash_check.equals(user) || hash_
check.equals(term) ||

    hash_check.equals(telep) || hash_check.equals(subsc) || hash_ch
eck.equals(spare) ||

    hash_check.equals(sound) || hash_check.equals(setProv) || hash_
check.equals(settings) ||

```

```

        hash_check.equals(sdk) || hash_check.equals(phone) || hash_check.equals(packIns) ||

        hash_check.equals(music) || hash_check.equals(mms) || hash_check.equals(media) ||

        hash_check.equals(launch) || hash_check.equals(latin) || hash_check.equals(imProv) ||

        hash_check.equals(html) || hash_check.equals(googSearch) || hash_check.equals(email) ||

        hash_check.equals(drmProv) || hash_check.equals(downProv) || hash_check.equals(dev) ||

        hash_check.equals(contProv) || hash_check.equals(contacts) || hash_check.equals(camera) ||

        hash_check.equals(calProv) || hash_check.equals(calendar) || hash_check.equals(calc) ||

        hash_check.equals(browser) || hash_check.equals(alarm) || hash_check.equals(framres) ||

        hash_check.equals(appinst) || hash_check.equals(algtut) || hash_check.equals(mictagread) ||

        hash_check.equals(tinyflash) || hash_check.equals(apricalc) || hash_check.equals(autokill) ||

        hash_check.equals(ligrac) || hash_check.equals(vidbox) )

        /* Match is found - indicate approved application */
        approvedContent = true;

        /* Flag containing approval result - 'true' if match found */
    }

    return approvedContent;
}

/* End of APS custom security function */

```

Bibliography

- [ARM05] ARM Architecture Reference Manual. Cambridge, England: ARM Limited, 2005.
- [BurE09] E. Burnette. *Hello, Android: Introducing Google's Mobile Development Platform*. Raleigh NC: The Pragmatic Bookshelf, 2009.
- [BurJ08] J. Burns. "Developing Secure Mobile Applications For Android: An Introduction to making secure Android applications". iSec Partners. October, 2008.
- [BurJ09] J. Burns. "Mobile Application Security on Android: Context on Android Security". iSec Partners. Prepared for Black Hat USA 2009. June, 2009.
- [Cha09] A. Chaudhuri. "Language-Based Security on Android". Programming Languages and Analysis for Security (PLAS) '09. Dublin Ireland. June 15, 2009.
- [DCI09] Dot Com Infoway. "Android by 2012: A study on present and future of Google's Android". Position Paper. <http://www.dotcominfoway.com>. October 2009.
- [Dim08] J. Dimarzio. *Android: A Programmer's Guide*. New York: The McGraw-Hill Companies, 2008.
- [EOM09] W. Enck, M. Ongtang, and P. McDaniel. "Understanding Android Security". IEEE Security & Privacy. 50-57. IEEE, January/February, 2009.
- [FCF10] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. "SCanDroid: Automated Security Certification of Android Applications". Submitted to IEEE S&P'10: Proceedings of the 31st IEEE Symposium on Security and Privacy. 2010.
- [FOG09] M. Feldmann, V. Oleniuk, L. Globa, and A. Schill. "Overview of a Model-to-Code Transformation Approach for Generating Service-Based Interactive Applications for Google Android". 2009 19th International Crimean Conference "Microwave & Telecommunication Technology". 362-364. Crimea Ukraine, September 14-18, 2009.
- [Gir10] J. Girard. "Isolated Android Attack Portends Future Exploits". Gartner Inc. ID Number: G00175285. March 12, 2010.

[GN10] Gartner Newsroom. “Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010; Smartphone Sales Increased 96 Percent”. <http://www.gartner.com/it/page.jsp?id=1466313>. Dec. 10, 2010.

[GT09] Good Technology. “Good for Enterprise – iPhone, Android and webOS”. Architecture and Security Whitepaper. <http://www.good.com>. 2009.

[Has08] C. Haseman. *Android Essentials*. Lexington KY: Apress firstPress, 2008.

[HK09] S. Y. Hashimi and S. Komatineni. *Pro Android*. Berkeley CA: Apress, 2009.

[JMH08] D. Jezard, J. Makkar, and D. Holding-Parsons. “Google Android”. Tiger Spike White Paper. Tiger Spike Limited, London. October 1, 2008.

[JTD09] M. Jantscher, M. Talhaoui, D. De Vos, I. Cunha, A. Roszcyk, and M. Datsyuk. “Android”. Android Team 2009. <http://www.mad-ip.eu/files/reports/Android.pdf>.

[Kim09] W. B. Kimball. *SecureQEMU: Emulation-Based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing*. MS thesis, AFIT/GCO/ENG/09-03. School of Engineering and Management, Air Force Institute of Technology (AU), Wright Patterson AFB OH, March 2009.

[LC09] N. Li and G. Chen. “Multi-Layered Friendship Modeling for Location-Based Mobile Social Networks”. Proceedings of the Sixth Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous) Toronto, Canada, 2009.

[MTV09] G. Macario, M. Torchiano, and M. Violante. “An In-Vehicle Infotainment Software Architecture Based on Google Android”. IEEE Symposium on Industrial Embedded Systems (SIES) 2009. 257-260. IEEE, 2009.

[Mur09] M. L. Murphy. *Beginning Android*. Berkeley CA: Apress, 2009.

[Nie93] J. Nielsen, *Usability Engineering*. San Francisco: Morgan Kaufmann, 1993.

[NW10] NielsenWire. “The State of Mobile Apps”. http://blog.nielsen.com/nielsenwire/online_mobile/the-state-of-mobile-apps/. Dec. 12, 2010.

[OME09] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. “Semantically Rich Application-Centric Security in Android”. Department of Computer Science

and Engineering, The Pennsylvania State University, University Park, PA. 2009.

- [PGT09] F. S. Park, C. Gangakhedkar, and P. Traynor. “Leveraging Cellular Infrastructure to Improve Fraud Prevention”. 2009 Annual Computer Security Applications Conference. 350-359. IEEE, 2009.
- [Phy10] PhysOrg. “How Secure are iPhone and Android Apps”. <http://www.physorg.com/news189356350.html>. April 1, 2010.
- [RML09] M. Raythattha, J. Moore, D. Lu, and S. Yang. “Google Android Strategy”. Google – Memo to the CEO. March 11, 2009.
- [SAH09] J. W. Stevenson, D. Albert, R. Hockauf. “Security and Trust on Android”. Proposal for Giesecke & Devrient Contribution. Giesecke & Devrient GmbH. November 4, 2009.
- [SBS09] A-D. Schmidt, R. Bye, H-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yüksel, S. A. Camtepe, and S. Albayrak. “Static Analysis of Executables for Collaborative Malware Detection on Android”. IEEE ICC 2009 Communication and Information Systems Security Symposium. Dresden, Germany, 2009.
- [SDT08] Spectrum Data Technologies. “Thoughts on Google Android”. A Spectrum White Paper. <http://www.spectrumdt.com>. February 2008.
- [SFK10] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. “Google Android: A Comprehensive Security Assessment”. Computing Now. IEEE Security & Privacy. IEEE Computer and Reliability Societies. vol. 8, no. 2, 35-44. March/April 2010.
- [SKF09] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. “Towards Formal Analysis of the Permission-based Security Model for Android”. icwmc, pp.87-92, 2009 Fifth International Conference on Wireless and Mobile Communications. 2009.
- [Tho09] C. Thomas. “Surviving in iPhone Territory”. The Conversation Group. Lessons in Competitive Brand – Building from the Android and G1 Phone Launch. 2009. <http://www.theconversationgroup.com/wp-content/uploads/2009/08/Google-Android-Social-Media-Landscape-Executive-Summary.pdf>.
- [UTG08] M. Ughetti, T. Trucco, and D. Gotta. “Development of agent-based, peer-to-peer mobile applications on ANDROID with JADE”. The Second

International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies. 287-294. IEEE Computer Society, 2008.

[YA09] C. Y. Yeon and S. M. Al-Marzouqi. "Practical Implementations for Securing VoIP Enabled Mobile Devices". 2009 Third International Conference on Network and System Security. 409-414. IEEE Computer Society, 2009.

[ZNA09] I. A. Zaulkernan, S. Nikkhah, and M. Al-Sabah. "A Lightweight Distributed Implementation of IMS LD on Google's Android Platform". 2009 Ninth IEEE International Conference on Advanced Learning Technologies. 59-63. IEEE Computer Society, 2009.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 24-03-2011			2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) Sep 2009 – Mar 2011	
4. TITLE AND SUBTITLE Android Protection Mechanism: A Signed Code Security Mechanism for Smartphone Applications			5a. CONTRACT NUMBER			
			5b. GRANT NUMBER			
			5c. PROGRAM ELEMENT NUMBER			
6. AUTHOR(S) Stueckle, Jonathan D., Capt, USAF			5d. PROJECT NUMBER			
			5e. TASK NUMBER			
			5f. WORK UNIT NUMBER			
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/11-06		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency, Intelligence Surveillance Reconnaissance PMO Attn: Dave Lehr, Military Investment Program Director 9800 Savage Road Ft. George G. Meade, MD 20755-6248 (240) 373-2548				10. SPONSOR/MONITOR'S ACRONYM(S) NSA OPS2B		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT This research develops the Android Protection System (APS), a hardware-implemented application security mechanism on Android smartphones. APS uses a hash-based white-list approach to protect mobile devices from unapproved application execution. Functional testing confirms this implementation allows approved content to execute on the mobile device while blocking unapproved content. Performance benchmarking shows system overhead during application installation increases linearly as the application package size increases. APS presents no noticeable performance degradation during application execution. The security mechanism degrades system performance only during application installation, when users expect delay. APS is implemented within the default Android application installation process. Applications are hashed prior to installation and compared against a white-list of approved content. APS allows applications that generate a matching hash; all others are blocked. APS blocks 100% of unapproved content while allowing 100% of approved content. Performance overhead for APS varies from 100.5% to 112.5% with respect to the default Android application installation process. This research directly supports the efforts of the USAF and the DoD to protect our information and ensure that adversaries do not gain access to our systems.						
15. SUBJECT TERMS Mobile device security, application security, signed code protection, android application security,						
16. SECURITY CLASSIFICATION OF:		17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Dr. Rusty O. Baldwin, ENG		
REPORT U	ABSTRACT U	c. THIS PAGE U	UU	19b. TELEPHONE NUMBER (Include area code) (937) 255-3636 x4445; Rusty.Baldwin@afit.edu		

Standard Form 298 (Rev: 8-98)

Prescribed by ANSI Std Z39-18